嵌入式 Linux 入门笔记

作者：阿南

前言

很多朋友（无论是单片机出身的底层软件工程师还是PC机的Windows程序员）都很想学习Linux（觉得只有嵌入式Linux才有钱途，而我觉得无论什么，只要比别人做的好，都会有前途），同时却觉得它很难，把它神密化了（就象当初神密化ARM一样），还认为必须要先买个ARM开发板才能开始Linux的学习，或问是否应先装个虚拟机等等，然后一直在徘徊着（时间就这样一天天过去了）！我讨厌徘徊，宁可花时间去尝试失败，然后总结经验。这份笔记基本上按时间顺序记录了我刚接触Linux到对它有个整体认识的学习、调试过程，出现的问题及心得总结等，也是我平时工作中不可缺少的手册。本想按内容重新整理，但后来还是保持着原来的时间顺序，因为觉得这样更容易让初学者借鉴学习过程和方法。Linux，我还只是一个新手，所以一定会有很多错误，很多非常不恰当的想法和问题，肯请见谅和批评指正！如果觉得它对您有帮助，甚至非常的棒，请您写信（ccn422@hotmail.com）告诉我，以给我最大的鼓励！如果您想表示感谢，那么请您上21IC BBS的ARM论坛，和我一起帮助那些需要帮助的朋友吧！我知道中国足球很差，但还会一如既往的关注和支持！自己也会执着的抽出时间，千方百计的寻找踢球机会，培养激情。同样，也知道中国的电子技术环境不好，但还会执着追求、学习下去，发表心得与大伙共勉。亲爱的工程师们，让我们象伊拉克足球一样战斗吧!!!

第一阶段 在PC机上学习熟悉Linux

刚学会ARM不久，就遇到很多工程师在学习和使用Linux，当时很是好奇和羡慕！注意到几乎所有工程师都拥有《GNU/Linux编程指南》，故也买来收藏着，看了些介绍性的篇幅，也有了在PC机上装个Linux系统的念头。由于对RedHat还不了解，包括基本的操作，于是买了本《Redhat9.0入门》，在电脑城找了张Readhat9.0光盘（现在肯定找不到了，去网上下吧），还担心在原来硬盘上安装会破坏原来的数据（担心是多余的，后来都用PartitionMagic在原盘符下直接分区），于是花了75元买了8.4G的旧硬盘，就这样回家瞎玩起来。后来又买了本《Redhat9.0系统管理》，应付RedHat9.0的基本操作已经足够。熟悉一两个星期的Redhat9.0基本操作与环境后，是该玩点深入的东东了，觉得自己将来应该是先以嵌入式Linux驱动为主，而不是应用编程，所以在还没有进一步学习《GNU/Linux编程指南》的情况下，就买《Linux设备驱动程序》钻研（后来证明这是错误的，有些急功近利）。《Linux设备驱动程序》看的比较费劲，通读了一遍后，在PC机做的第二个实验就遇到了困难，编译总是出错。去书店参考其它驱动的书，上网查找等试了很多方法都没有解决，困惑了很久。后来又开始研读《GNU/Linux编程指南》，读了这本书，再翻《Linux设备驱动程序》就轻松多了。问题没有解决总会有个结，会时常有针对性的去书店翻些相关的书，上网查些资料等。在了解到了内核源码树结构，编译等之后，才知道驱动和应用程序是有区别的，是属于内核级，在编译时要指定Linux内核源代码树下的头文件（－I/usr/src/linux-2.4/include），问题就这样解决，以后的学习、实验都变得顺利了，出现问题基本都能很快排除。总结这阶段的学习顺序，我觉得应该是：首先，在PC机上安装Linux系统，再买本相应的入门书籍，主要是熟悉Linux环境，学习常用的命令和操

作（不一定多，基本、常用的就可以，以后在使用过程中慢慢积累），理解各目录结构与作用等。其次，学习在 Linux 环境下编程，《GNU/Linux 编程指南》就可，它会介绍文件描述符的概念，打开、读、写等操作的系列基础知识，没有这些基础而直接看《Linux 设备驱动程序》会觉得累。后来研读了《UNIX 环境高级编程》，觉得也很好，它讲了很多前者没有的细节。再次，拿本内核的书翻翻，了解一下 linux 内核源代码树的目录结构，编译等。最后学习《Linux 设备驱动程序》，理解驱动程序的结构框架等。注：我觉得学习不需要都直接记住，有点不现实（但应该理解，不要留下疑问，如果有，应及时的用实验去证实再记录），在以后的应用中再查阅巩固，这阶段也不例外。

一. Red Hat Linux 9 下的常用操作

1. 如何修改在开机引导装载程序中，等待自动登录默认操作系统的时间？

答：如果引导装载程序是 GRUB，则修改/etc/grub.conf 文件中的 timeout=秒数。

如果引导装载程序是 LILO, 则修改的是/etc/lilo.conf 文件。可用 vi 等编辑器修改,下同。

2. 在字符(Text)模式下，如何关机、重启、注销？

答：关机：poweroff 或 shutdown –h now；重启：reboot 或 shutdown –r now；注销（即重新登入）：logout；其中在 shutdown 指令中的 now 是指现在就执行，也可以指定多少时间后再执行此命令。

3. U 盘的使用

答：先创建/mnt/usb 目录，再执行 mount /dev/sda1 /mnt/usb 挂载，此时/mnt/usb 就是 U 盘的目录，在拔出 U 盘时要执行 umount /mnt/usb 进行卸载。

4. 在字符模式(Text)下，如何进入 X Window 模式（Graphic）？在 X Window 模式下，如何返回字符模式？

答：执行 startx 命令 启动 X Window 模式；鼠标点击“Main Menu(主菜单)->Log out(注销)”打开对话框中，选择“注销”进入字符模式；或 CRTL+ALT+F1~F6 来进入不同的虚拟控制台（即文本模式下）。

5. 如何重新指定开机默认进入的执行模式（字符或 X Window 模式）？

答：修改/etc/inittab 文件中的内容（id:5:initdefault:)。

其中，5 表示以 X Window 模式（Graphic）登入，3 为字符模式(Text)登入

6. 在字符模式下，如何使用户登入时，系统不要求输入密码？如何恢复或更改用户密码？

答：取消输入密码：passwd –d 用户帐号。如要取消root登入时的密码，则执行passwd –d root。也可以用vi打开/etc/shadow文件，删除密码的方法取消。恢复或更改密码则执行passwd 用户帐号(如果是取消自己则不用)命令后会提示输入New password和Retype new password。

7. 字符模式下，如何新增用户帐号？

答：使用“useradd 用户帐号” 命令来增加，但在新增后还不能登入使用，还需要用 passwd 命令来设置密码后才行。

8. 在 X Window 下，如何选择系统默认使用的语言？

答：鼠标点击“主菜单->系统设置->语言”打开选择语言对话框中选择。

9. 用 ls 等命令查看的内容太多，超过一页时，如何分页显示？

答：可用 ls | more 或 ls | less 进行分页查看。其中，在用 more 浏览时，按空格键（Space）则会显示下一页的内容；按回车（Enter）键则会向下多显示一行；按 q 键则离开浏览模式。

在用 less 浏览时，按 h 键会出现在线使用说明；按 q 键离开浏览模式。

10. 如何获得命令的使用方法？

答：可利用在线手册—man(Manual)，用法是输入 man 和待查的命令名称。如要查询 ls 命令的使用方法，则输入以下命令：man ls 。也可以 ls --help

11. 搜索文件及目录和搜索包含特定字符串的文件？

答：搜索文件及目录可以用 find 命令，如要在根目录（/）上搜索 apache 文件则输入命令：find / -name apache –print,注意：如果没有指定目录，则系统会以当前的目录为搜索的范围；搜索包含特定字符串的文件可以用 grep 命令，如要在/etc 目录下搜索包含字符串“password”

的文件则输入：grep －n ‘password’ /etc/*.*，其中加入-n 参数会标出符合指定的字符串的列数，另外不可指定在目录中搜索，否则会出现错误信息，如上述不能写成：grep －n ‘password’ /etc/
另外如果想停止搜索可以直接按“Ctrl＋C”键结束该命令就可以。现在我常用：grep －ir password /etc
12. 控制台间的切换
答：在文本模式下，用 ALT+F1~F6 来分别在 6 个虚拟控制台间切换，它们可分别用不同的用户名登入和执行不同的命令与程序，如果已经启动了 X Window(如:在文本模式下用”startx”命令启动)，则按 ALT+F7 切换到 X Window 图形模式。
在 X Window 图形模式下，用 CRTL+ALT+F1~F6 分别切换到文本模式下的 6 个虚拟控制台。CRTL + ALT + BackSpace 结束图形模式。
因为 linux 是多任务的系统,所以可以在不同的控制台下用不同（或同一）的用户登陆来运行不同的程序。我觉得这个功能很方便，因为有时在文本模式下，需要打开多个终端来处理显示多个的任务，如：一个终端运行 minicom 作为目标板的控制，一个终端作为宿主机编译目标板要运行的文件，还有多个终端打开多个源文件在浏览等等。如果习惯在 X Window 模式下就例外，因为用鼠标右键就可以打开多个终端。
13. 查看 PDF 文档和浏览网页?
答：在 X Window 下打开 shell 终端，输入”xpdf filename.pdf”和“mizzo filename.html” 命令分别查看。注：必须在 X Window 下才能运行这两个程序，文本模式不能运行。
14. 查看磁盘使用情况
答：#df －h
15. /proc 目录下，几个关于系统资源非常有用的文件
/proc/modules、/proc/ioports、/proc/iomen、/proc/devices、/proc/interrupts、/proc/filesystems
16. 关于内核代码调试时输出打印信息的 printk 语句
如：printk(KERN_DEBUG “Here I am : %s :%i\n”,__FILE__, __LINE_&_)；
printk(KERN_INFO “ Driver Initional \n”)；等同于 printk(“<6>” “ Driver Initional \n”)；
printk(“<1> Hello, World!\n”)；
没有指定优先级的 printk 语句采用默认日志级别(DEFAULT_MESSAGE_LOGLEVEL)在 kernel/printk.c 中被指定，根据日志级别，内核可能会把消息输出到当前控制台上。当优先级小于 console_loglevel 整数值时，消息才会被显示出来。如果系统同时运行了 klogd 和 syslogd，则无论 console_loglevel 为何值，都将把内核消息追加到/val/log/messages 中。console_loglevel 的初始值是 DEFAULT_CONSOLE_LOGLEVEL，可以通过文本文件/proc/sys/kernel/printk 来读取和修改它及控制台的当前日志级别等。也可以简单的输入下面命令使所有的内核消息得到显示：
#echo 8 > /porc/sys/kernel/printk
17. 查看当前正在运行的进程
答：#ps
18. 解压缩到指定目录
答：#tar xvzf linutte.tgz －C /linuette
19. 当/etc/grub.conf 文件中的内容被修改或破坏时不能正常启动时，如何在 GRUB 引导时修改设置使其正常启动
答：以修改了/etc/grub.conf 文件中的 vga 项使启动时显示器不能显示为例，在 GRUB 启动引导菜单中 windows XP 和 Red Hat Linux(2.4.20-8)两项中使用键头键选中 linux 系统，不按[Enter]，而按[E]键进入菜单项目编辑器，再使用键头键选中 kernel 项，也按[E]键进行编辑，在行的后面输入 vga=791 fb=on 后按[Enter]，最后按[b]键执行命令，并引导操作系统。
20. 包管理器 RPM 使用，以 tmake 为例
安装：#rpm –ivh tmake-1.7-3mz.noarch.rpm

升级：#rpm –Uvh tmake-1.7-3mz.noarch.rpm
查询：#rpm –q tmake
删除：#rpm –e tmake

二. Minicom的使用

1. 启动Minicom
   输入minicom启动，或输入minicom －s直接进入设置模式。
2. 设置
   1> 选择串口：在选择菜单中的“Serial port setup”，按回车，再按“A”以设置“Serial Device”（如果使用串口1，则输入/dev/ttyS0，如果您使用串口2，则输入/dev/ttyS1，注意其中的S是大写），按回车返回。
   2> 设置波特率：按“E”键进入设置“bps/par/Bits”（波特率）界面，如果按“I”以设置波特率为115200，按回车返回。
   3> 数据流控制：按“F”键设置“Hardware Flow Control”为“NO”。
   其它为缺省设置 ，然后按回车到串口设置主菜单，，选择“Save setup as dfl”，按回车键保存刚才的设置（保存到“/etc/minirc.dfl），再选择“Exit”退出设置模式，回到minicom操作模式。
   此时可像Windows下的超级终端一样使用了。
3. 退出minicom
   按下“Ctrl+A”键，松开后紧接着再按下“Q”键,在跳出的窗口中，选择“Yes”。
4. 其它有用的功能

命令帮助 -- “Ctrl+A”后按 “ Z ”
清屏 -- “Ctrl+A”后按 “ C ”
设置 -- “Ctrl+A”后按 “ O ”
发送文件 -- “Ctrl+A”后按 “ S ”
退出 -- “Ctrl+A”后按 “ Q ”

三. NFS的使用

为什么要使用NFS：网络文件系统（NFS，Network File System）是一种在网络上的计算机间共享文件的方法，通过它可以将计算机上的文件系统导出给另一台计算机。我们在宿主机上编辑、编译好的程序，可以通过它导出到目标板上进行实际的运行。

1. 宿主机配置
   从NFS服务器中共享文件又称导出目录，/etc/exports文件控制NFS服务器要导出哪些目录。格式如下：

共享的目录 可以连接的主机 (读写权限，其它参数)

如果允许目标板（IP：192.168.0.*）挂载主机的/home目录，则/etc/exports文件的内容如下：

/home 192.168.*.*(rw，sync)

注：如果出现mount不成功，可将sync去掉试试

更改后要使用如下命令重新载入配置文件：

#/sbin/services nfs reload 或 #/etc/init.d/nfs reload

然后启动NFS服务器，命令如下：

/sbin/services nfs start

上面两个命令也可以用下面的一条指令完成，如下：

/sbin/service nfs restart

设置好后也可以通过mount自己来测试NFS服务设置是否成功。如果本机IP为192.168.0.1,则可以用mount 192.168.0.1:/home /mnt，如果mount成功，则在/mnt的目录就可以看到/home目录下面的东西了。

2. 使用mount命令挂载NFS文件系统

下面将宿主机（IP：192.168.0.1）配置的/home目录挂载到（IP：192.168.0.7）目标板上的/mnt目录。

在宿主机启动minicom作为目标板的显示终端，启动目标板的linux系统，再使用下面命令：

mount －o nolock 192.168.0.1:/home /mnt

注意：如果没有“-o nolock”选项，而直接使用命令：mount 192.168.0.1:/home /mnt 时将出现如下错误：portmap：server localhost not responding，timed out

目前我都用：mount －t nfs 192.168.0.1:/home /mnt

四．应用程序编程实验

《GNU/Linux 编程指南》一书的例程是非常全面的，下载它的源代码后，可以直接编译运行，基本上不会有什么问题。

五．模块编程实验

参考《Linux设备驱动程序》P25页，hello.c源程序如下：

```
#define MODULE
#include <linux/module.h>
int init_module(void){
     printk (“<1>Hello,world\n”);
     return 0;
}
void cleanup_module(void){
     printk(“<1>Goodbye cruel world\n”);
}
```

用以下命令进行编译：

```
#gcc －c hello.c
#insmod ./hello.o
hello,world
#rmmod hello            //注：不是#rmmod hell.o
Goodbye cruel world
```

出现的问题：在执行#insmod ./hello.o 时并没出打印出“hello,world”而是出现了下述错误：./hello:kernel －module version mismatch

hello.o was compiled for kernel version 2.4.20

while this kernel is version 2.4.20-8

原因：模块和内核版本不匹配，即编译内核的编译器与现在编译模块的编译器版本不一致。

解决方法：1> 将/usr/include/linux/version.h 文件中的#define… “2.4.20”修改成#define…“2.4.20-8”(),再重新编译!

2> 用insmod的-f（force,强制）选项强行装入模块，如下：insmod －f ./hello.o

3> 因用vi /usr/include/linux/version.h查看到定义的内核版本是2.4.20，而在内核原代码树下/usr/src/linux-2.4/include/linux/version.h中定义为2.4.20-8版本，所以用如下命令进行编译：

gcc －c －I/usr/src/linux-2.4/include hello.c，再装载就 OK！

注：可查看/proc/modules 文件。

六．简单的字符设备驱动实验

驱动程序源代码如下：

```
/*CharDriver.c*/
#define  _NO_VERSION
#include <linux/module.h>
#include <linux/version.h>
char kernel_version[] = UTS_RELEASE;
#define  KERNEL
#include <linux/types.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/errno.h>
#include <asm/segment.h>
#define  SUCCESS 0

static int device_read(struct file *file, char *buf, size_t count, loff_t *f_pos);
static int device_open(struct inode *inode, struct file *file);
static void device_release(struct inode *inode, struct file *file);

struct file_operations tdd_fops = {
    read: device_read,
    open: device_open,
    release: device_release,
};
#define DEVICE_NAME "char_dev"
static int Device_Open = 0;
unsigned int test_major = 0;
//static char Messaege[1024];
static int device_open(struct inode *inode, struct file *file){
    #ifdef DEBUG
    printk ("device_open(%p)\n", file);
    #endif
    if (Device_Open)
        return -EBUSY;
    Device_Open++;
    MOD_INC_USE_COUNT;
    return SUCCESS;
}

static void device_release(struct inode *inode, struct file *file){
#ifdef DEBUG
```

```
    printk ("device_release(%p,%p)\n", inode, file);
#endif
    Device_Open--;
    MOD_DEC_USE_COUNT;
}

static int device_read(struct file *file, char *buf, size_t count, loff_t *f_pos){
    int left;
    if (verify_area(VERIFY_WRITE, buf, count) == -EFAULT)
        return -EFAULT;
    for (left = count; left > 0; left--){
        put_user(1, buf);
        buf++;
    }
    return count;
}

int init_module(void){
    int result;
    result = register_chrdev(0, "char_dev", &tdd_fops);
    if (result < 0){
        printk("char_dev:can't get major number\n");
        return result;
    }
    if (test_major == 0)
        test_major = result;
    printk ("Hello,I'm in kenel mode\n");
    return 0;
}

void cleanup_module(void){
    printk("Hello,I'm goint to out\n");
    unregister_chrdev(test_major,"char_dev");
}
```

用下述命令进行编译：

```
#gcc －O2 -DMODULE －D__KERNEL__ -c CharDriver.c
```

出现下述错误：

```
CharDriver.c 18：Tvariable ‘fops’ has initializer but incomplete type
CharDriver.c 19：unknown field ‘read’ specified in initializer
CharDriver.c 20：unknown field ‘open’ specified in initializer
CharDriver.c 20：unknown field ‘release’ specified in initializer
CharDriver.c 53：storage size of ‘fops’ isn’t known
```

原因：在用 gcc 进行编译时，默认搜索的 include 文件路径为/usr/include，但/usr/include/linux/fs.h 中没有定义 file_operations 结构体（可以打开该文件看一下），file_operations 是在原代码树目录

/usr/src/linux-2.4/include 下的 linux/fs.h 中定义，而且用了#ifdef __KERNEL__条件编译。因此在编译时必须指定该目录和定义__KERNEL__。编译命令如下：

#gcc －O2 -DMODULE －D__KERNEL__ －I/usr/src/linux-2.4/include -c CharDriver.c

经过编译得到的文件 CharDriver.o 就是设备驱动程序，执行以下命令把它安装到系统中：

#insmod CharDriver.o

如果安装成功，用#vi /proc/devices 打开文件将看到设备 char_dev 和主设备号。如果需要卸载，可运行

#rmmod char_dev

然后就需要创建设备文件了，执行以下命令：mknod /dev/char_dev c 254 0，其中“/dev/char_dev”是要生成的设备名及目录，选项 c 是指字符设备，254 是生成的主设备号，可用上面方法查看到，也可能是 253 或其它，从设备号设成 0 就可以。另外可以用 rm －f /dev/char_dev 将删除 mknod 命令创建的设备文件。可用下述测试程序源代码进行测试。

驱动的测试程序如下：

```
/*DriverTest.c*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main(int argc, char *argv[]){
    int testdev;
    int i;
    char buf[10];
    testdev = open("/dev/char_dev", O_RDWR);
    if (testdev == -1){
        printf("Cann't open file \n");
        exit(0);
    }
    read(testdev, buf, 10);
    for (i = 0; i < 10; i++){
        printf("%d\n", buf[i]);
    }
    close(testdev);
}
```

需要注意的是将驱动程序数据传给用户程序时是 put_user(1,buf)，而不是 put_user(1,buf，1);否则会出现错误。还有 copy_to_user(buf,Buffer++,1)函数的 Buffer 是指针变量，而不是具体的数据，如果写成 copy_to_user(buf,1,1)，也会出错。

第二阶段 在开发板上学习研究 Linux

公司虽然用 ARM 已经多年，但都是裸奔，产品在功能和性能上的提升都出现了瓶颈，急需软件平台的突破，这也增大了我的学习动力。过年假期没有回家（尽管妈妈生病刚动完手术，但还是含着泪在电话里告诉爸爸我的决定），而是抓紧时间专心研究基于 s3c2410 的嵌入式 Linux 平台。硬件是一块普普通通的 s3c2410 开发板，软件选择 MIZI Linux SDK (Software Development Kit) for S3C2410，因为我觉得它是 s3c2410 最原始的平台，国内大部分的开发板都是在它的基础上发展过来，选择它比较可靠，心里也比较

踏实，以后应用于产品也是如此。从 mizi ftp 站点下载的 SDK 光盘，包含了开发板原理图、BOM 表、linux 开发平台所需的所有软件、工具及使用手册等，资料是比较齐全的，完全可以参考它设计出自己的开发板。下面的软件安装、工具使用等内容摘自和总结于它的操作手册，LED、按键驱动实验的源代码摘自广州友善之臂科技有限公司的 SBC-2410X-UM.pdf。

一. MIZI Linux SDK for S3C2410 开发环境及工具使用

1. 构造软件开发环境

1．1 安装基本的软件开发工具

安装 RedHat 9 的工作站(workstation)模式，当完成后基本的开发工具也就安装完成，主要有：binutils 、gcc 、gcc-c++ 、glibc-kernheaders 、glibc-common 、glibc 、glibc-devel 、patch 、make 、minicom 等。可以用 rpm 包管理程序来查看它们的版本信息，如查看 binutils 的命令如下：

#rpm －q binutils

1．2 安装 MIZI Linux SDK (Software Development Kit) for S3C2410

先从 mizi ftp 站点下载光盘的镜像文件，再刻录成光盘（如果直接解压会有一些小问题出现）。

将光盘放入光驱，然后执行下述命令：

```
#mkdir /linuette
#mount /dev/cdrom /mnt/cdrom
#cd /mnt/cdrom
#tar cvf /linuetter/linuetter.tar *
#cd /linuette
#tar xvf linuetter.tar
#rm －rf linuetter.tar
```

其实上述的操作就是将光盘里的所有内容复制到 /linuette 目录下，但为何不直接用 cp 命令去实现呢？是因为 cp 命令会改变文件一些属性的原因吗？及 cp 时会出现一些不能创建 ln 等错误。

另外，MIZI 文档中的”#tar cf -.|(cd /linuette ; tar xvf -)” 命令为何会出错?

1．3 安装交叉编译环境

进入 RPMS 目录下执行包管理程序安装所有的*.rpm 程序。注：不能直接用一条“#rpm －Uvh *.rpm”命令进行安装就完成，这样没有用！必须一步步的安装，如下：

Tool chain

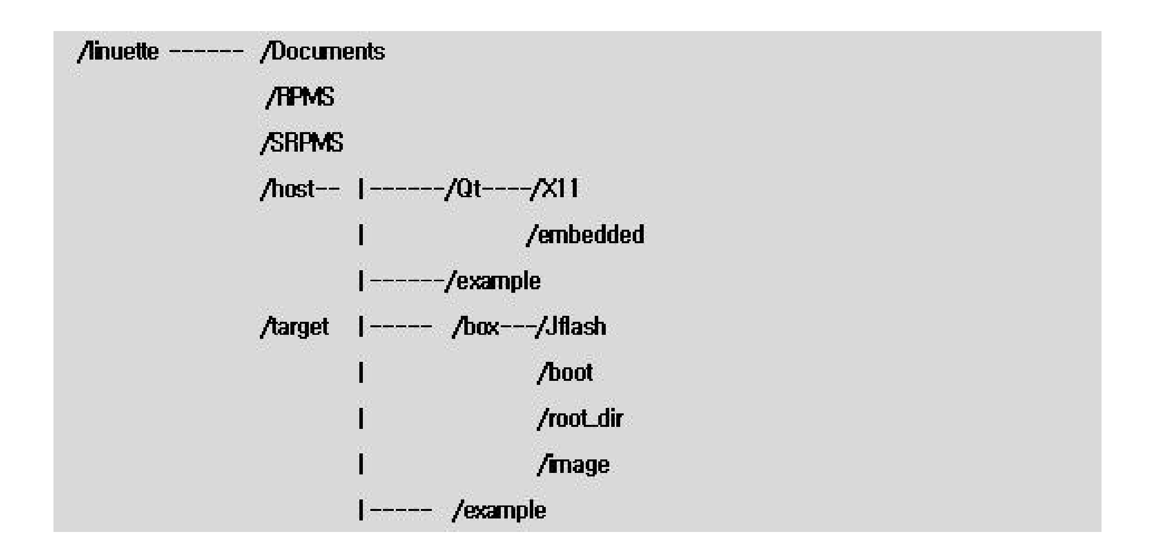
```
#rpm －Uvh cross－armv4l－binutils－2.10－3mz.i386.rpm
#rpm －Uvh cross－armv4l－kernel－headers－2.4.5_rmk7_np2－1mz.i386.rpm
#rpm －Uvh cross－armv4l－gcc－2.95.2－10mz.i386.rpm
#rpm －Uvh cross－armv4l－glibc－2.2.1－2mz.i386.rpm
#rpm －Uvh cross－armv4l－gcc－c++－2.95.2－10mz.i386.rpm
```

MIZI-tool-box

```
#rpm －Uvh cross－armv4l－libfloat－1.0－3mz.i386.rpm
```

#rpm -Uvh cross-armv4l-zlib-1.1.3-5mz.i386.rpm
#rpm -Uvh cross-armv4l-jpeg-6b-2mz.i386.rpm
#rpm -Uvh cross-armv4l-jpeg-devel-6b-2mz.i386.rpm
#rpm -Uvh tmake-1.7-3mz.noarch.rpm
GUI Toolkit
#rpm -Uvh linuette_sdk_x86-1.5-3mz.noarch.rpm
#rpm -Uvh qtE-custom-1.5-1mz.i386.rpm
#rpm -Uvh linuette_sdk_arm-1.5-1mz.noarch.rpm
#rpm -Uvh cross-armv4l-qtE-custom-1.5-4mz.i386.rpm
可以将上述写成一个脚本或直接将安装完成的目录打包，下次安装就比较方便了。

1．4 MIZI Linux SDK 的目录结构



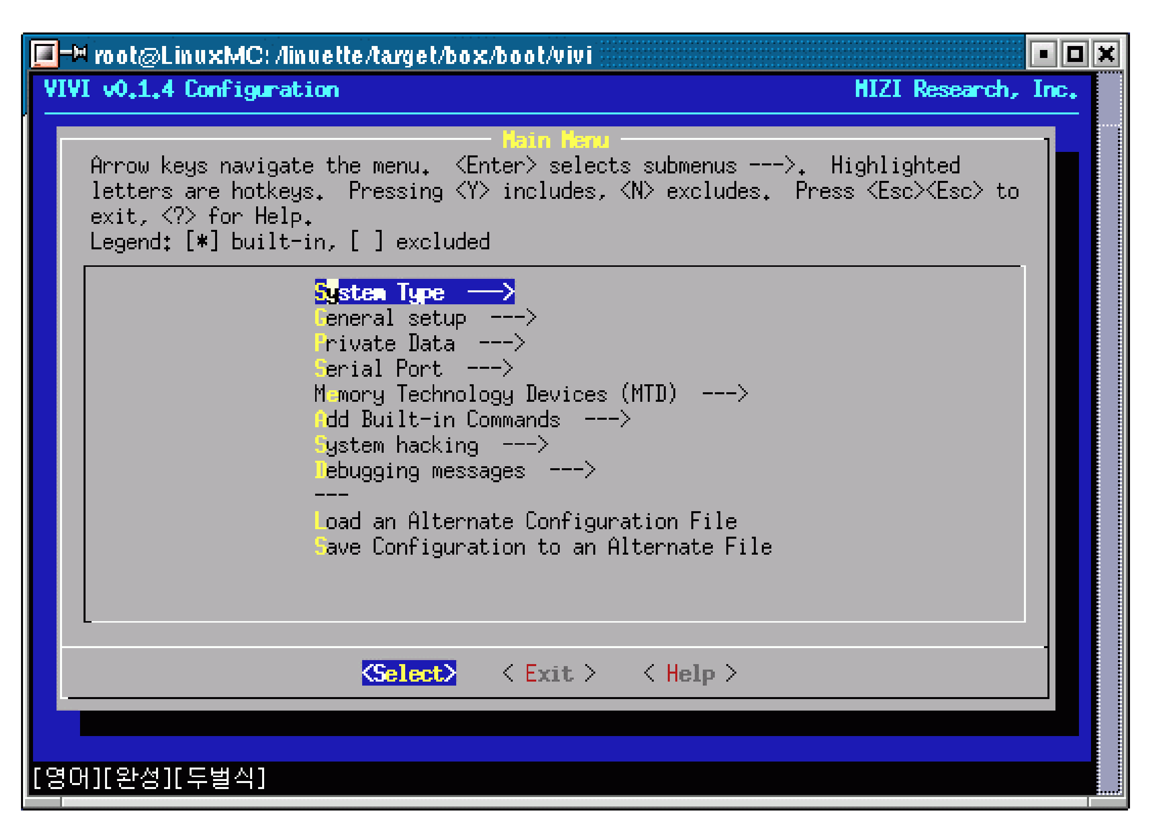
2. 编译嵌入式 Linux 生成 image

嵌入式 Linux 软件系统由 bootloader、kernel、root filesysytem 和可选的 user filesystem 构成，如下：

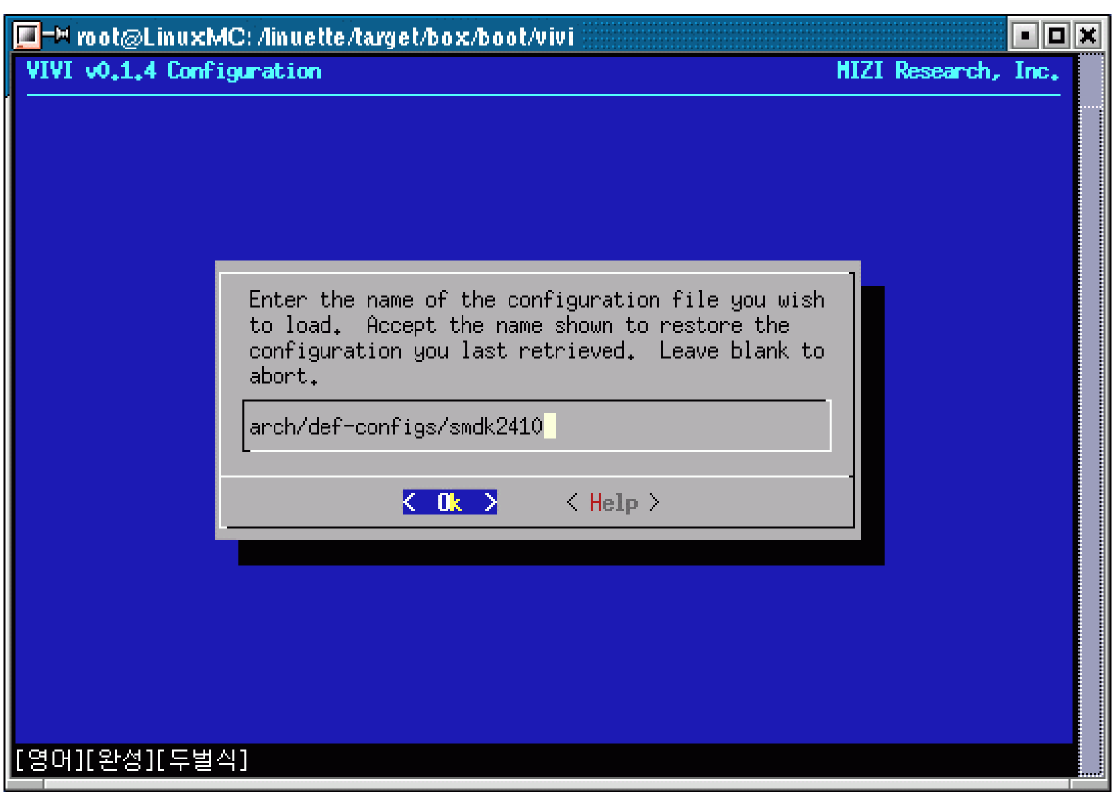
bootloader（一次固化）＋ 内核（多次更新）＋ 根文件系统（多次更新） [+ user filesystem]

2．1 配置编译 bootloader —— vivi

#cd /linuette/target/box/boot
#tar jxvf vivi.tar.bz2（如果压缩文件的后缀是.gz 则用该命令：#tar zxvf vivi.tar.gz）
#cd vivi
#make menuconfig



选择“Load an Alternate Configuration File”菜单，然后写入“arch/def-configs/smdk2410”。



再选择“OK”，最后选择“Yes”退出。

#make

如果不出错，则在目录：/linuette/target/box/boot/vivi/下将生成 vivi 的二进制文件。

#mkdir /image

#cp vivi /image

2．2 配置编译内核

```
#cd /linuette/target/box
#tar jxvf linux-2.4.18-rmk7-pxa1-mz4.tar.bz2
#cd kernel
#make menuconfig
```

与编译 vivi 类似，在“Load an Alternate Configuration File”菜单后写入“arch/arm/def-configs/smdk2410”。

退出后执行：

编译内核

```
#make zImage
```

编译模块

```
#make modules
#make modules_install
#cp zImage /image
```

将模块打包，准备更新文件系统模块

```
#cd /tmp/lib/modules/2.4.18-rmk7-pxa1
#tar zcvf mod.tgz kernel pcmcia
```

2．3 生成根文件系统

2．3．1 生成不包括 GUI 的根文件系统

```
#cd /linuette/target/box/root_dir
#tar jxvf root.tar.bz2
#./mkcramfs ./root ./root.cramfs
#cp root.cramfs /image
```

2．3．2 生成包括 GUI 的根文件系统

```
#cd /linuette/target/box/root_dir
#tar jxvf root_english.tar.bz2 (或者 root_ hangul.tar.bz2 和 root_china.tar.bz2)
#cd root_english/usr/lib/modules/2.4.18-rmk7-pxa1
#rm -rf kernel/ pcmcia/
#tar zxvf /tmp/lib/modules/2.4.18-rmk7-pxa1/mod.tgz  -C .
#./mkcramfs ./root_english ./root_english.cramfs
#cp root_english.cramfs /image
```

3. 将嵌入式 Linux 的 image 下载到目标板

3．1 第一次(或系统不能启动时)下载 image

3.1.1 使用 JTAG 接口下载 vivi

用JTAG小板通过PC机的并口与目标板的JTAG 接口将它们相连。

```
#cd /linuette/target/box/Jflash
#cp ./Jflash-s3c2410 /image
```

#cd /image

#./Jflash-s3c2410 --help

根据目标板的Nand Flash选择相应项，如:K9S1208

#./Jflash-s3c2410 vivi /t=5

Select the function to test : 0

Input target block number: 0

Select the function to test : 2

3.1.2 通过 vivi 下载各 images

3.1.2.1 进入 vivi 的下载模式

由于 vivi 正常工作在启动加载模式，启动等待时间可能很短，所以要想进入下载模式，必须先按着空格键，再复位目标板。

#cd /image

#minicom

Supply power to target board, then press “space-bar” quickly.

vivi>

已进入下载模式，输入“help”将列出所有可用的命令。

3.1.2.2 Flash 分区（Partitioning）

注：如果没有这一步，而直接下载内核和根文件系统时，系统可能无法正确引导，会出现下述错误：

Kenel panic：VFS：Unable to mount root fs on 61:02

所以必须进行分区，再下载 bootload、kernel、root filesystem。

vivi>bon part 0 192k 1M

分区后存储器分配如下：

0 ~ 192K :   vivi (bootload)

192K ~ 1M :   zImage (kernel)

1M ~ End-part: root.cramfs (root filesystem)

3.1.2.3 下载 vivi image

vivi>load flash vivi x

快速按下“Ctrl + A” 和 “S”,再选择 xmodem 发送模式

切换到/image 目录，光标移到 vivi 后按空格选中，再按回车开始发送。

如果发送失败，原因是执行装载命令到xmodem_initial的时间太短，而产生超时。可以将“xmodem_initial_timeout”参数加大。

#param show

……

#param set xmodem_initial_timeout 5000000:“5000000”means 5 second because a unit is μs.

#param save

修改完后，再次执行上述操作。

3.1.2.4 下载 kernel image

与下载 vivi 同样的操作方法来下载 kernel image —— zImage

vivi>load flash kernel x

3.1.2.5 下载 root-filesystem image

与下载 vivi 同样的操作方法来下载 root-filesystem —— root.cramfs

vivi>load flash root x

当都下载完毕后

vivi>boot

如果正确将引导进入 linux 的 shell 模式。

3．2 系统能正常引导时下载 image

当能够正常引导进入操作系统时，可以先通过以太网将 image 文件和 imagewrite 烧写工具复制到目标板，再在目标板上执行烧写，这样速度会更快（以太网的速度要比串口快很多，不要像我一样傻，用 xmodem 下载带 GUI 的文件系统，当你等了半个多小时后出错，定会气的吐血）。

将所需要下载到目标板的文件都放入/image 目录下，以方便传送

#cd /linuette/target/box/image/

#cp ./imagewrite /image

3．2．1 ztelnet

ztelnet 与 redhat9 提供的 telnet 不同的是在登录后支持文件的发送命令，即：

ztelnet>sz filename [filename]

注：在 RPMS 目录下有它的安装包，要先安装。

用“imagewrite”写 flash 的过程是在目标系统上进行的，所以要在宿主机上通过 ztelnet 登录到目标机，再把宿主机上/image 目录的内容发送到目标机，再执行 imagewrite 进行烧写。使用交叉网线连接宿主机与目标机，再在宿主机的 Linux 系统中启动两个终端窗口，一个运行 minicom 作为目标机的控制台，另一个在/image 目录下。

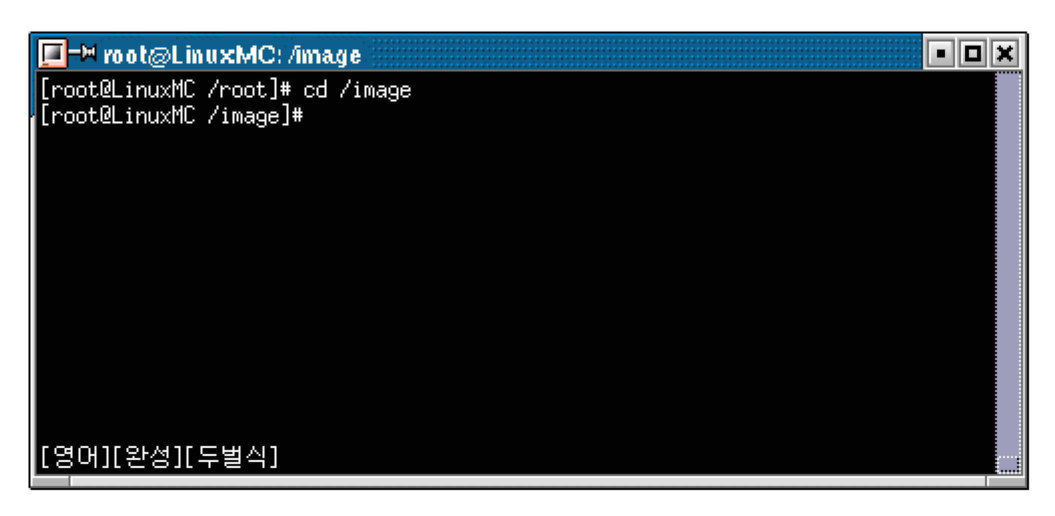
Terminal 1：terminal which location is image directory

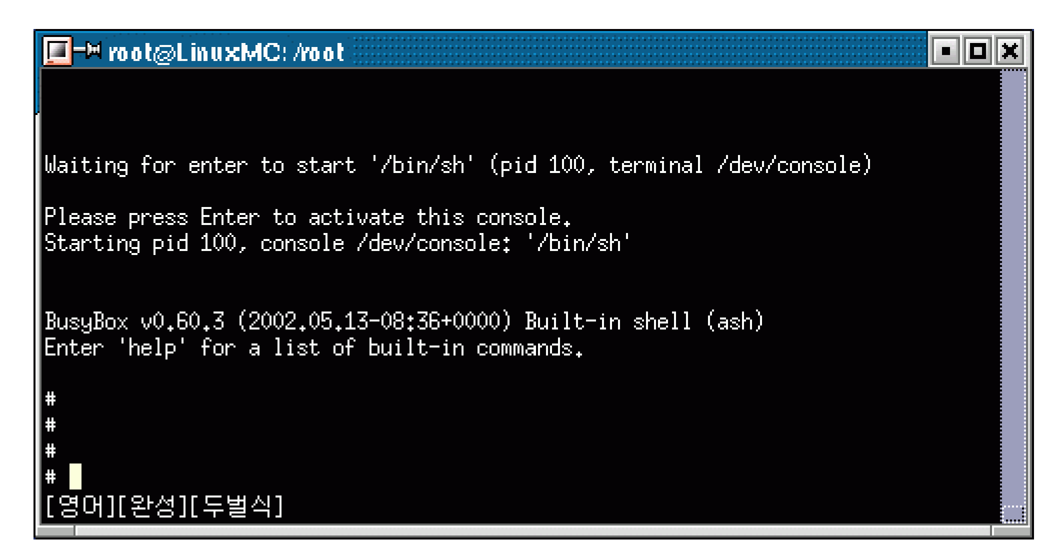
Terminal 2：terminal which executes minicom (console of target board)

启动目标机，使其也进入 shell 模式下，为双机通信作好准备。

Terminal 1：#cd /image

Terminal 2：#minicom





设置双方的 IP 地址

Terminal 1：#ifconfig eth0 down

#ifconfig eth0 10.10.10.1:Set up arbitrary IP.

Terminal 2：#ifconfig eth0 10.10.10.2:Set up IP that can make a pair with that of host PC.

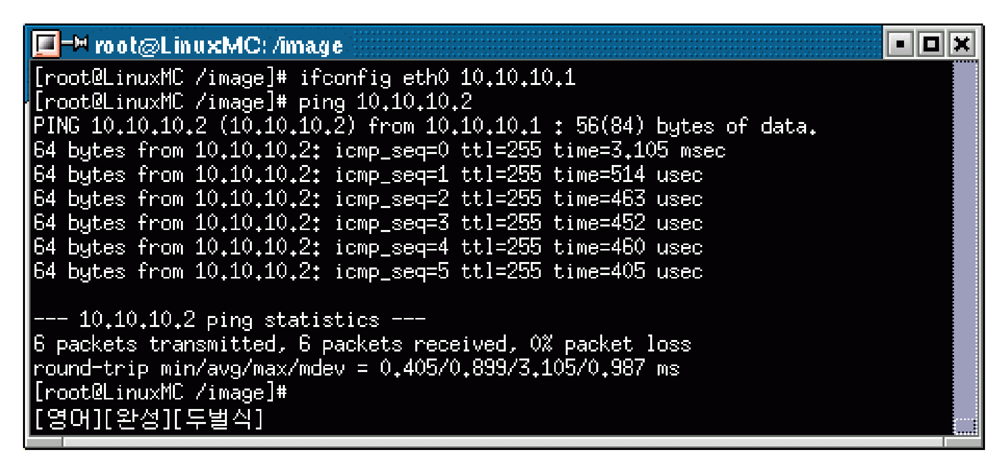
#inetd

inetd 命令用于启动 telnet 服务器

检测双方能否常通信

Terminal 1： #ping 10.10.10.2

: We can confirm that it’ s communicating between host PC and target board by ping command.



宿主机用 ztelnet 登录目标机

Terminal 1： #ztelnet 10.10.10.2

Login by root account. Because you don’ t need to input password, press “Enter” key.

```
root@LinuxMC: /image
64 bytes from 10.10.10.2: icmp_seq=5 ttl=255 time=405 usec

--- 10.10.10.2 ping statistics ---
6 packets transmitted, 6 packets received, 0% packet loss
round-trip min/avg/max/mdev = 0.405/0.899/3.105/0.987 ms
[root@LinuxMC /image]# ztelnet 10.10.10.2
Trying 10.10.10.2...
Connected to 10.10.10.2.
Escape character is '^]'.

Linux 2.4.18-rmk7-pxa1 ((none)) (0)


(none) login: root
[영어][완성][두벌식]
```

发送 images

只有/tmp 目录可以被双方读写，其它的都是只读文件系统，所以必须下载到该目录下。

Terminal 1:

#cd /tmp

#rz

Pushing “Ctrl +]”， “ztelnet> ” console appears.

telnet>sz vivi zImage root.cramfs imagewrite

```
root@LinuxMC: /image
# rz
rz ready. To begin transfer, type "sz file ..." to your modem program

┬┬B0100000023be50
ztelnet> sz vivi zImage root_hangul.cramfs imagewrite
Retry 0: Awaiting pathname nak for vivi
Retry 1: Awaiting pathname nak for zImage
 623616 ZMODEM CRC-32    Retry 1: Awaiting pathname nak for root_hangul.cramfs
28155904 ZMODEM CRC-32    Retry 1: Awaiting pathname nak for imagewrite
   8192 ZMODEM CRC-32    sz: File skipped by receiver request
File skipped by receiver request
sz 3.25 2-11-95 finished.
#
#
#
#
[영어][완성][두벌식]
```

可以用“ls”命令来检查下载的内容

```
root@LinuxMC: /image
B0100000023be50
ztelnet> sz vivi zImage root_hangul.cramfs imagewrite
Retry 0: Awaiting pathname nak for vivi
Retry 1: Awaiting pathname nak for zImage
 623616 ZMODEM CRC-32    Retry 1: Awaiting pathname nak for root_hangul.cramfs
28155904 ZMODEM CRC-32    Retry 1: Awaiting pathname nak for imagewrite
   8192 ZMODEM CRC-32      sz: File skipped by receiver request
File skipped by receiver request
sz 3.25 2-11-95 finished.
#
#
#
# ls
imagewrite          root_hangul.cramfs  zImage
qtembedded-unknown  vivi
#
[영어][완성][두벌식]
```

由于是cramfs文件系统，因此当电源掉电后，所有下载的文件都会消失。如果想保存必须将其写入到flash中。

3．2．2 用 imagewrite 将 images 写入到 flash

3．2．2．1 Flash 分区

与上一节一样，将给flash分三个区，分别存放bootload、kernel和root filesystem。

#./imagewrite /dev/mtd/0 -part 0 192K 1M

```
root@LinuxMC: /image
#
#
# ./imagewrite /dev/mtd/0 -part 0 192K 1M
meminfo size = 67108864
doing partition
size = 0
size = 196608
size = 1048576
check bad block
part = 0 end = 196608
part = 1 end = 1048576
part = 2 end = 67108864
1544000: is bad
3344000: is bad
k = 0 block = 1297
k = 1 block = 3217
[영어][완성][두벌식]
```

3．2．2．2 将 images 写入 flash

格式：#./imagewrite <mtd_dev> <file:offset>

将 vivi 写入到“0~192KB”区，zImage 写入到“192KB~1MB”，root.cramfs 写入到“1MB~End-part”。

#./imagewrite /dev/mtd/0 vivi:0

#./imagewrite /dev/mtd/0 zImage:192K

#./imagewrite /dev/mtd/0 root.cramfs:1M

3．3 出现过的问题

3．3．1 下载完 vivi、kernel、root 后执行 boot 时，出现下述现象后就停止了，而不能正常引导进入 linux 系统。

NOW, Booting Linux......

Uncompressing Linux... done, booting the kernel.

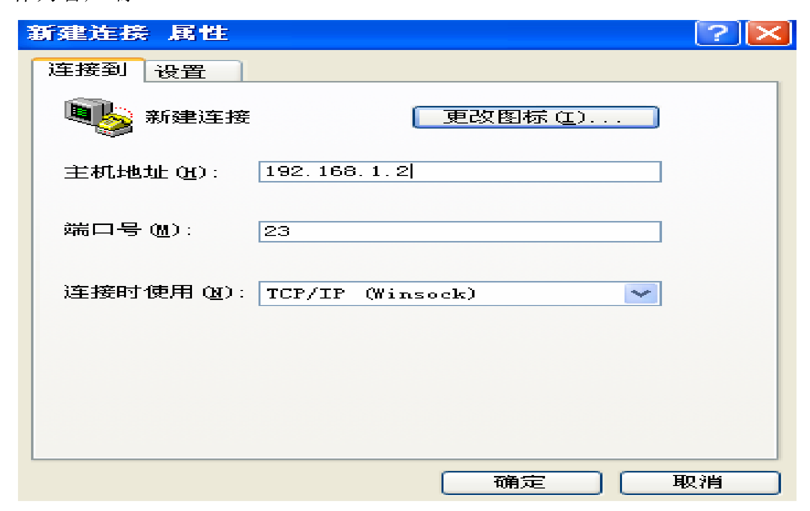
问题：可能是硬件接触不良，或内核下载有些问题，如没有按规定分区，重新分区下载试试。

3．3．2 登录到 telnet 后不能发送文件，即不支持 sz 命令。

RedHat 9 自带的 telnet 不支持该功能，需安装光盘里的 ztelnet，使用 ztelnet 登录。

3．4 windows 下发送文件到 linux 目录

可以由 windows 自带的超级终端，或 SecureCRT 完成。在 Windows 操作系统下选择“开始”->“程序”->“附件”->“通讯”->“超级终端”，打开超级终端，选择“文件”->“属性”命令，在“连接时使用”项中选择“TCP/IP（winsock)”，在“主机地址”项中输入目标板的 IP 地址（如下图所示），将目标板作为服务器，PC 作为客户端。



设置完成后，点击“呼叫”，在超级终端上将出现目标板的 linux 系统的 telnet 登录界面，可用 root 登录，如下：

Linux 2.4.18-rmk7-pxa1 ((none)) (0)

(none) login: root

BusyBox v0.60.3 (2002.05.13-08:36+0000) Built-in shell (ash)

Enter 'help' for a list of built-in commands.

#

切换到/tmp 目录下后选择“传送”->“发送文件”，点击“浏览”选中要发送的文件后点“发送”按钮，如下：

为 11 发送 Zmodem 与崩溃恢复 文件

正在发送：E:\soft_copy\ARM\2410\zImage

上次事件：正在发送　　文件：1 / 1

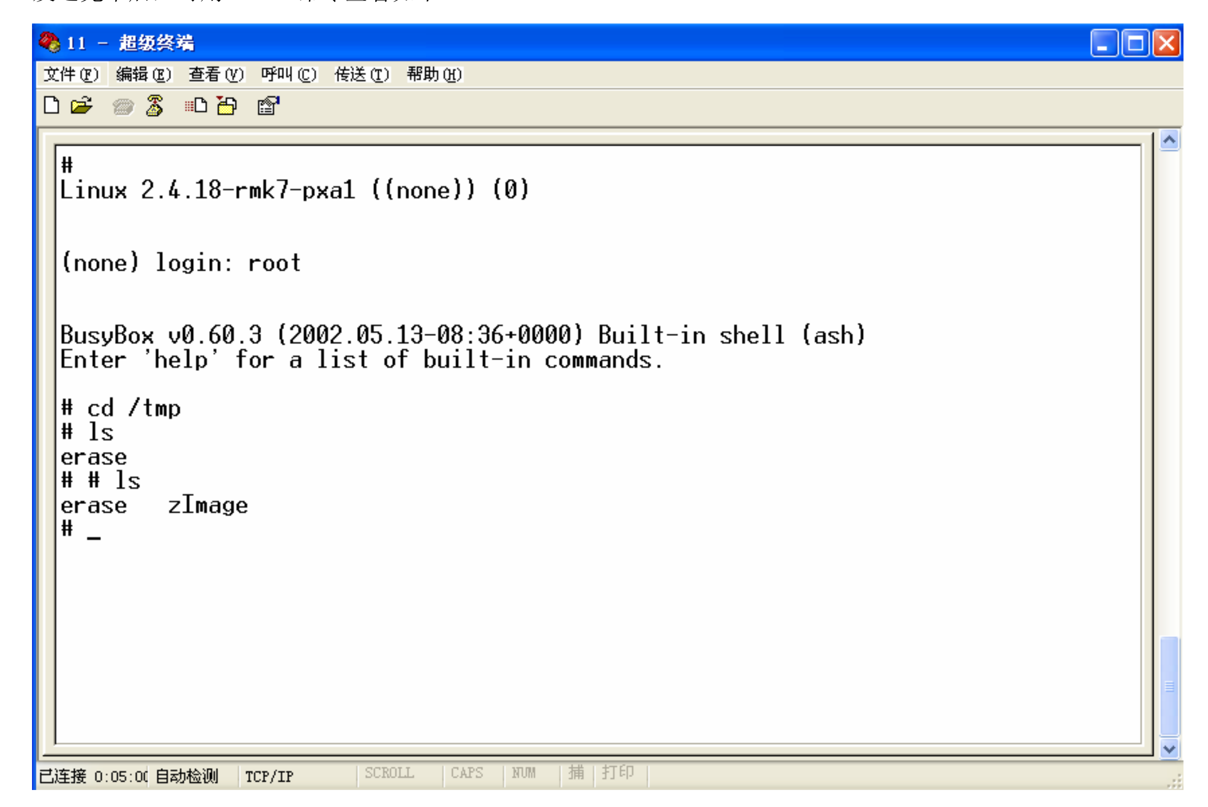
状态：正在发送　　重试次数：18

文件：　　209K / 707K

已用：00:00:03　剩余：00:00:05　吞吐量：92874 cps

取消　cps/bps (C)

发送完毕后，可用“ls”命令查看如下：



4. 嵌入式平台测试

我们为目标板编译、运行最简单的 hello 程序，以测试、熟悉上述的开发环境及应用开发。

4．1 hello 源程序

程序清单：

```
/*hello.c - 在终端上打印 hello,linux*/
#include <stdio.c>
int main(void){
```

```
    printf(“hello,linux\n”);
    return 0;
}
```

4．2 Makefile

可以直接输入编译命令，也可以写 Makefile 脚本进行编译。

脚本清单:

```
CROSS=/opt/host/armv4l/bin/armv4l-unknown-linux-
all: hello
hello:
    $(CROSS)gcc -o hello hello.c
clean:
    rm －rf hello *.o
```

第一行是指定交叉编译器的安装路径

#make

生成 hello 可执行文件

4．3 程序运行

当生成执行文件后，可用上节提到的 ztelnet 方法将其下载到目标板的/tmp 目录下，再执行:

#./hello

二．嵌入式 Linux 驱动开发

1．模块编程实验

1．1 源程序

模块程序清单:

```
#define MODULE
#include <linux/module.h>
int init_module(void){
    printk (“<1>Hello,world! \n”);
    return 0;
}
void cleanup_module(void){
    printk(“<1>Goodbye cruel world! \n”);
}
```

1．2 PC 机上编译运行

```
#gcc  －c hello.c
#insmod ./hello.o
hello,worlk!
#rmmod hello
Goodbye cruel world!
#
```

1．3 为目标板上编译和运行

Makefile 脚本清单:

```
CROSS=/opt/host/armv4l/bin/armv4l-unknown-linux-
INCPATH=/linuette/target/box/kernel/include
all: hello
hello:
    $(CROSS)c  －I$(INCPATH) hello.c
clean:
    rm  －rf *.o
```

写好脚本后，输入 make 编译

```
#make
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc  －I/linuette/target/box/kernel/include
```

将生成的目标文件下载到目标板的/tmp 目录下，再动态装载和卸载时将在终端打印相应的字符串，如下：

```
#insmod ./hello.o
Hello,world!
#rmmod hello
Goodbye cruel world!
```

当装载后，可以查看 /proc/modules 的内容，是否装载成功。

```
#cat /proc/modules
```

2. 点亮目标板的 LED

2．1 LED 驱动

2．1．1 LED 驱动源程序清单

```
#ifndef __KERNEL__
#define __KERNEL__
#endif
#ifndef MODULE
#define MODULE
#endif
#include <linux/module.h>
```

```
#include <linux/kernel.h>
#include <linux/version.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <asm-arm/arch-s3c2410/hardware.h>
#define DEVICE_NAME "leds"
#define LED_MAJOR 232

static unsigned long led_table[]={GPIO_B7,GPIO_B8,GPIO_B9,GPIO_B10};
static int leds_ioctl(struct inode *inode,struct file *file,unsigned int cmd,
                      unsigned long arg){
    switch(cmd){
        case 0:
        case 1:
                if (arg > 4){
                    return -EINVAL;
                }
                write_gpio_bit(led_table[arg],!cmd);
        default:
                return -EINVAL;
    }
}
static struct file_operations leds_fops={
    owner:THIS_MODULE,
    ioctl:leds_ioctl,
};
static int __init leds_init(void){
    int ret;
    int i;
    ret = register_chrdev(LED_MAJOR,DEVICE_NAME,&leds_fops);
    if (ret < 0){
        printk(DEVICE_NAME "can't register major number");
        return ret;
    }
    for (i=0;i<4;i++){
        set_gpio_ctrl(led_table[i] | GPIO_PULLUP_EN | GPIO_MODE_OUT);
        write_gpio_bit(led_table[i],1);
    }
    printk(DEVICE_NAME "initialized\n");
    return 0;
}
static void __exit leds_exit(void){
    unregister_chrdev(LED_MAJOR,DEVICE_NAME);
}
```

```
module_init(leds_init);
module_exit(leds_exit);
```

2．1．2 Makefile

```
CROSS=/opt/host/armv4l/bin/armv4l-unknown-linux-
INC=/linuette/target/box/kernel/include/
all: leds
leds:
 $(CROSS)gcc -O2 -c -I$(INC) leds.c
clean:
 rm -rf leds *.o
```

2．1．3 驱动的动态装载

装载模块

#insmod leds.o

如果装载成功，在/proc/devices 文件中可以看到 leds 和它的主设备号(232)。

#cat /proc/devices

创建设备文件

#mknod /dev/leds c 243 0

成功后在/dev 目录下将生成 leds 设备文件

#ls /dev

2．2 LED 驱动的测试

2．2．1 测试程序 1 － 单个 LED 的控制

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
int main(int argc,char **argv){
    int on;
    int led_no;
    int fd;
    if (argc != 3 || sscanf(argv[1],"%d",&led_no) != 1 || sscanf(argv[2],"%d",&on) != 1 ||
on < 0 || on > 1 || led_no < 0 || led_no > 3){
        fprintf(stderr,"Usage: leds led_no 1|0\n");
        exit(1);
    }
    fd = open("/dev/leds",0);
    if (fd < 0){
        perror("open device leds");
       exit(1);
    }
    ioctl(fd,on,led_no);
    close(fd);
    return 0;
```

}

2．2．2 Makefile

```
CROSS=/opt/host/armv4l/bin/armv4l-unknown-linux-
all: leds
leds:
 $(CROSS)gcc －o led_test leds.c
clean:
 rm －rf led_test *.o
```

2．2．3 测试

分别控制四个 LED 亮、灭

```
#./leds_test 0 1
#./leds_test 1 1
#./leds_test 2 1
#./leds_test 3 1
#./leds_test 0 0
#./leds_test 1 0
#./leds_test 2 0
#./leds_test 3 0
```

2．3 将驱动编译进内核

当在实际的项目的量产中，不可能一台台的将一堆驱动一个个的安装，一定要将它们编译进内核，而且当为 CRAMFS 文件系统时，一掉电，刚安装的模块及驱动就丢了，所以也要求必须编译进内核。

2．3．1 在内核配置选项中增加该驱动的选项

#cd /linuette/target/box/kernel

#vi ./drivers/char/Config.in

增加内容：

```
if [ “$CONFIG_ARCH_S3C2410” = “y” ]; then
    tristate ‘S3C2410 LED Driver example’ CONFIG_S3C2410_LED
    或直接只用该行
fi
```

#make menuconfig

选中“Character devices->S3C2410 LED Driver example”项。

2．3．2 在内核的 Makefile 中增加该驱动的编译内容

#vi ./driver/char/Makefile

增加内容：

obj-$(CONFIG_S3C2410_LED) += leds.o

将 led.c 复制到 kernel/driver/char 目录下，然后重新编译更新内核。

2．3．3 出现的问题

问题 1：make zImage 后出现__this_modules 无定义的错误

原因：在驱动程序中有下述的定义项

```
#ifndef MODULE
#define MODULE
#endif
```

问题 2：不能编译进内核，即/proc/devices 文件和/dev 目录下都没有出 leds

原因：在驱动程序中有下述的定义项

```
#ifndef MODULE
#define MODULE
#endif
```

问题 3：在/dev 目录下找不到 leds 设备文件

解决方法：可以使用设备文件系统 devfs。将原驱动程序中的初始化和退出程序修改成如下：

```
static devfs_handle_t devfs_handle;
static int __init leds_init(void){
    int ret;
    int i;
    ret = register_chrdev(LED_MAJOR,DEVICE_NAME,&leds_fops);
    if (ret < 0){
        printk(DEVICE_NAME "can't register major number");
        return ret;
    }
    devfs_handle = devfs_register(NULL,DEVICE_NAME,DEVFS_FL_DEFAULT,LED_MAJOR,0,S_IFCHR |
S_IRUSR | S_IWUSR,&leds_fops,NULL);
    for (i=0;i<4;i++){
        set_gpio_ctrl(led_table[i] | GPIO_PULLUP_EN | GPIO_MODE_OUT);
        write_gpio_bit(led_table[i],1);
    }
    printk(DEVICE_NAME "initialized\n");
    return 0;
}
static void __exit leds_exit(void){
    devfs_unregister(devfs_handle);
    unregister_chrdev(LED_MAJOR,DEVICE_NAME);
}
```

2．4 系统启动后，自动使 LED 闪烁

最终的程序肯定是不会象上述一样需要输入命令才执行，而是当系统启动后就自动的一直在运行。

2．4．1 LED 测试程序 2 -- LED 循环计数

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
int main(void){
    unsigned int on;
    int led_no;
    int fd;
    unsigned char i;
    fd = open("/dev/leds",0);
```

```
    if (fd < 0){
        perror("open device leds");
        exit(1);
    }
    on = 0;
    while (1) {
        for (i = 0; i < 4; i++){
            ioctl(fd,(on>>i)&1,i);
        }
        on += 1;
        on &= 0xf;
        sleep(1);
    }
    close(fd);
    return 0;
}
```

2．4．2 Makefile

```
CROSS=/opt/host/armv4l/bin/armv4l-unknown-linux-
all: leds
leds:
 $(CROSS)gcc  -o led_inc leds.c
clean:
 rm  -rf led_inc *.o
```

编译生成 leds_inc 可执行文件，通过 ztelnet 下载到目标板的/tmp 目录再运行就可以看到 4 个 LED 在不断的计数显示。

2．4．3 启动系统时自动运行该程序

可以先把可执行文件 leds_inc 复制到根文件系统 root 目录下的 sbin 目录，再在系统的启动脚本中执行该程序，该脚本为根文件系统 root 目录下的 linuxrc 文件。假设现在 leds_inc 所在的目录下。

```
#cp leds_inc /linuette/target/box/root/sbin
#cd /linuette/target/box/root
#vi linuxrc
```

在 exec /bin/sh 的前面增加下面行

```
/sbin/leds_inc
```

也可以当执行该程序时，往终端打印提示信息，则再加上

```
echo “/sbin/leds_inc”
```

将修改过的根文件系统重新生成 image 文件

```
#cd ..
#./mkcramfs ./root ./root.cramfs
```

更新目标板的根文件系统，再启动。

3．按键中断实验

Linux 内核维护了一个中断信号线的注册表，模块在使用中断前要先申请一个中断通道(或者中断请求 IRQ)，然后在使用后释放该通道。下列在头文件<linux/sched.h>中声明的函数实现了该接口：

```
int request_irq(unsigned int irq,
              void (*handler)(int, void *, struct pt_regs *),
              unsigned long flag,
              const char *dev_name,
              void *dev_id);
void free_irq(unsigned int irq, void *dev_id);
```

3.1 按键驱动源程序

```
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/miscdevice.h>
#include <linux/sched.h>
#include <linux/delay.h>
#include <linux/poll.h>
#include <linux/spinlock.h>
#include <linux/irq.h>
#include <linux/delay.h>
#include <asm/hardware.h>
#define DEVICE_NAME    "custom-buttons"       //定义按键设备名
#define BUTTON_MAJOR   232                    //定义按键主设备号
static struct key_info {
     int irq_no;
     unsigned int gpio_port;
     int key_no;
} key_info_tab[4] = {                         //按键所使用的CPU资源
     { IRQ_EINT1, GPIO_F1, 1 },
     { IRQ_EINT2, GPIO_F2, 2 },
     { IRQ_EINT3, GPIO_F3, 3 },
     { IRQ_EINT7, GPIO_F7, 4 },
};
static int ready = 0;
static int key_value = 0;
static DECLARE_WAIT_QUEUE_HEAD(buttons_wait); //声明、初始化等待队列buttons_wait
static void buttons_irq(int irq, void *dev_id, struct pt_regs *reg)//中断处理程序
{
     struct key_info *k;
     int i;
     int found = 0;
     int up;
     int flags;
     for (i = 0; i < sizeof key_info_tab / sizeof key_info_tab[1]; i++){
```

```
        k = key_info_tab + i;
        if (k->irq_no == irq) {
            found = 1;
            break;
        }
    }
    if (!found){
        printk("bad irq %d in button\n", irq);
        return;
    }
    save_flags(flags);
    cli();                                    //禁用所有中断
    set_gpio_mode_user(k->gpio_port, GPIO_MODE_IN);
    up = read_gpio_bit(k->gpio_port);
    set_external_irq(k->irq_no, EXT_BOTH_EDGES, GPIO_PULLUP_DIS);
    restore_flags(flags);
       if (up) {
             key_value = k->key_no + 0x80;
       } else {
             key_value = k->key_no;
       }
        ready = 1;
    wake_up_interruptible(&buttons_wait);     //唤醒 buttons_wait 等待对列中的进程
}
static int request_irqs(void){
//申请系统中断，中断方式为双边延触发，即在上升沿和下降沿均发生中断
    struct key_info *k;
    int i;
    for (i = 0; i < sizeof key_info_tab / sizeof key_info_tab[1]; i++){
        k = key_info_tab + i;
        set_external_irq(k->irq_no, EXT_BOTH_EDGES, GPIO_PULLUP_DIS);// 双边触发
        if (request_irq(k->irq_no, &buttons_irq, SA_INTERRUPT, DEVICE_NAME, &buttons_irq)){
        //0 表示成功，负值表示出错
            return -1;
        }
    }
    return 0;
}
static void free_irqs(void)                   //释放中断
{
    struct key_info *k;
    int i;
    for (i = 0; i < sizeof key_info_tab / sizeof key_info_tab[1]; i++){
        k = key_info_tab + i;
```

```
        free_irq(k->irq_no, buttons_irq);
    }
}
static int matrix4_buttons_read(struct file * file, char * buffer, size_t count, loff_t *ppos)
{                                       //*file_operations 的"读"指针函数实现
    static int key;
    int flags;
    int repeat;
       if (!ready)
              return -EAGAIN;
       if (count != sizeof key_value)
              return -EINVAL;
    save_flags(flags);
    if (key != key_value){
        key = key_value;
        repeat = 0;
    }
    else{
        repeat = 1;
    }
    restore_flags(flags);
    if (repeat){
        return -EAGAIN;
    }
    copy_to_user(buffer, &key, sizeof key);   //使用 copy_to_user 把键值送到用户空间
    ready = 0;
    return sizeof key_value;
}
static unsigned int matrix4_buttons_select(struct file *file, struct poll_table_struct *wait){
    if (ready)
        return 1;                           //POLLIN，设备可以无阻塞读
    poll_wait(file, &buttons_wait, wait);     //把当前进程放入一个等待对列
    return 0;
    //使进程调用 poll 或 select 系统调用时，阻塞直到 buttons_wait 等待队列唤醒
}
static int matrix4_buttons_ioctl(struct inode *inode, struct file *file, unsigned int cmd,
unsigned long arg){
    switch (cmd){
    default:
        return -EINVAL;
    }
}
static struct file_operations matrix4_buttons_fops = {
    owner:   THIS_MODULE,
```

```
    ioctl:  matrix4_buttons_ioctl,
    poll: matrix4_buttons_select,
/*poll 方法是 poll 和 select 这两个系统调用的后端实现,是在用户空间程序执行 poll 或 select 系统调用时被调用,这两个系统调用可用来查询设备是否可读或可写，或是否处理某种特殊状态。这两个系统调用是可阻塞的，直至设备变为可读或可写状态为止*/
    read: matrix4_buttons_read,
};
static devfs_handle_t devfs_handle;
static int __init matrix4_buttons_init(void){  //按键初始化
    int ret;
    ready = 0;
    ret = register_chrdev(BUTTON_MAJOR, DEVICE_NAME, &matrix4_buttons_fops);//注册按键设备
    if (ret < 0){
      printk(DEVICE_NAME " can't register major number\n");
      return ret;
    }
    ret = request_irqs();                      //请求中断
    if (ret){
        unregister_chrdev(BUTTON_MAJOR, DEVICE_NAME);
        printk(DEVICE_NAME " can't request irqs\n");
        return ret;
    }
    devfs_handle = devfs_register(NULL, DEVICE_NAME, DEVFS_FL_DEFAULT,
                BUTTON_MAJOR, 0, S_IFCHR | S_IRUSR | S_IWUSR, &matrix4_buttons_fops, NULL);
                //使用 devfs 进行注册
    return 0;
}
static void __exit matrix4_buttons_exit(void){
    devfs_unregister(devfs_handle);
    free_irqs();
    unregister_chrdev(BUTTON_MAJOR, DEVICE_NAME);
}
module_init(matrix4_buttons_init);
module_exit(matrix4_buttons_exit);
MODULE_LICENSE("GPL");
```

3.2 按键中断测试程序

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```c
#include <fcntl.h>
#include <sys/select.h>
#include <sys/time.h>
#include <errno.h>
int main(void){
    int buttons_fd;
    int key_value;
    buttons_fd = open("/dev/buttons", 0);
    if (buttons_fd < 0) {
        perror("open device buttons");
        exit(1);
    }
    for (;;){
        fd_set rds;
        int ret;
        FD_ZERO(&rds);
        FD_SET(buttons_fd, &rds);
        ret = select(buttons_fd + 1, &rds, NULL, NULL, NULL);//阻塞直到可读
        if (ret < 0){
            perror("select");
            exit(1);
        }
        if (ret == 0){
            printf("Timeout.\n");
        }
        else if (FD_ISSET(buttons_fd, &rds)){
            int ret = read(buttons_fd, &key_value, sizeof key_value);
            if (ret != sizeof key_value){
                if (errno != EAGAIN)
                    perror("read buttons\n");
                continue;
            }
            else{
                printf("buttons_value: %d\n", key_value);
            }
        }
    }
    close(buttons_fd);
    return 0;
}
```

4．定时器驱动及 PWM 输出

s3c2410 分别有 Timer 0，1，2，3，4 共 5 个定时器，其中 Timer4 作为 linux 的系统调度时钟，Timer3

被DMA驱动所使用，故只有Timer 0，1，2共3个定时器可用，它们都具有PWM输出功能。

4．1 驱动功能描述

1> 从TOUT0输出38KHz，占空比50%的PWM波，且具有中断功能，在中断服务程序中判断是否停止本定时器来关闭PWM输出，实现调制功能。

2> 能够使能、禁止PWM输出

3> 有PWM输出波形数量的记录功能，且可随时读取，清零

4．2 PWM驱动程序

4．2．1 模块入口函数：module_init(pwm_init);和module_exit(pwm_cleanup);

pwm_init函数主要完成注册字符设备驱动程序、注册设备文件。pwm_cleanup函数将先前注册的都释放掉。初始化PWM使用的输出口。

```
static devfs_handle_t devfs_handle;
static int __init pwm_init(void){
    int  result;
    result = register_chrdev(PWM_MAJOR,  DEVICE_NAME,  &pwm_fops);
    if (result < 0){
        printk(KERN_ERR DEVICE_NAME ": Unable to get major %d\n", PWM_MAJOR);
        return(result);
    }
    devfs_handle = devfs_register(NULL, DEVICE_NAME, DEVFS_FL_DEFAULT,
          PWM_MAJOR, 0, S_IFCHR | S_IRUSR | S_IWUSR, &pwm_fops, NULL);
set_gpio_ctrl(GPIO_B0 | GPIO_PULLUP_EN | GPIO_MODE_TOUT);//TOUT0
    printk(KERN_INFO DEVICE_NAME ": init OK\n");
    return(0);
}
static void __exit pwm_cleanup(void){
    devfs_unregister(devfs_handle);
    unregister_chrdev(PWM_MAJOR, DEVICE_NAME);
}
module_init(pwm_init);
module_exit(pwm_cleanup);
MODULE_LICENSE("GPL");
```

4．2．2 pwm_open

主要功能：注册PWM中断处理函数，设置PWM的输出频率freq和占空比rate。其中freq由TCNTB0寄存器值（div）和定时器的输入时钟决定，定时器的输入时钟由TCFG0寄存器的Prescaler0值和TCFG1的MUX0（divider value）决定。公式如下：

Timer input clock Frequency = PCLK / {prescaler value+1} / {divider value}

Freq = Timer input clock Frequency / div

取PCLK = 50，prescaler value = 1，divider value = 2，要使Freq = 38KHz，则

div = Timer input clock Frequency / Freq = PCLK / {prescaler value+1} / {divider value} /Freq = 50MHz /(1+1) / 2 / 38KHz = 328.947 = 329

即 TCNTB0 = div = 329，当 rate = 50%时，TCMPB0 = 329 / 2 = 164

另外 CPU 的 FCLK、HCLK、PCLK 可由 kernel/arch/arm/mach-s3c2410/cpu.c 文件下的函数读取，在系统启动过种中打印出来。如下：

```
freq = s3c2410_get_cpu_clk();
 hclk = s3c2410_get_bus_clk(GET_HCLK);
 pclk = s3c2410_get_bus_clk(GET_PCLK);
CPU clock = 200.000 Mhz, HCLK = 100.000 Mhz, PCLK = 50.000 Mhz
```

s3c2410.h 中关于定时器的一些寄存器定义说明

```
TCFG0 = (TCFG0_DZONE(0) | TCFG0_PRE1(15) | TCFG0_PRE0(0));
#define TCFG0_DZONE(x)       FInsrt((x), fTCFG0_DZONE)
#define fTCFG0_DZONE         Fld(8,16)
#define Fld(Size, Shft) (((Size) << 16) + (Shft))
```

则 fTCFG0_DZONE 为((8 << 16) + 16)

TCFG0_DZONE(x) 为 FInsrt((x), ((8 << 16) + 16))

```
#define FInsrt(Value, Field)      (UData (Value) << FShft (Field))
#ifndef __ASSEMBLY__
#define UData(Data) ((unsigned long) (Data))
#else
#define UData(Data) (Data)
#endif
#define FShft(Field)    ((Field) & 0x0000FFFF)

TCFG0_DZONE(x) 为 (unsigned long)x << (((8 << 16) + 16) & 0x0000FFFF)
TCFG0_DZONE(x) 为 (unsigned long)x << 16
TCFG0_PRE1(x)  为  (unsigned long)x << (((8 << 16) + 8) & 0x0000FFFF)
TCFG0_PRE1(x)  为  (unsigned long)x << 8
TCFG0_PRE0(x)  为  (unsigned long)x << (((8 << 16) + 0) & 0x0000FFFF)
TCFG0_PRE0(x)  为  (unsigned long)x << 0
```

源程序：

```
static int pwm_open(struct inode *inode, struct file *filp){
    unsigned long flag;
    if (usage == 0){
        request_irq(IRQ_TIMER0, pwm_irq_handle, SA_INTERRUPT, "my" DEVICE_NAME, NULL);
        local_irq_save(flag);
        TCFG0 = (TCFG0 & 0xFFFF00) | 1;  //prescaler value=1
        TCFG1 = (TCFG1 & 0xFFFFF0) | 0;  //divider value=1/2
        TCNTB0 = 329;           //38KHz
        TCMPB0 = 164;           //50%
        TCON = (TCON & 0x7FFFE0) | 0x09;//Timer0:auto reload;Update TCNTB0,TCMPB0;start
        RunTime = 0;
        local_irq_restore(flag);
    }
    usage++;
    MOD_INC_USE_COUNT;
```

```
    return 0;                    // success
}
```

4. 2. 3 pwm_release

主要是将 PWM 中断释放，及停止定时器运行及 PWM 输出，源程序：

```
static int pwm_release(struct inode *inode, struct file *filp) {
    unsigned long flag;
    MOD_DEC_USE_COUNT;
    usage--;
    if (usage == 0){
        local_irq_save(flag);
        TCON = (TCON & 0x7FFFE0) | 0x0A;//Timer0:auto reload;Update TCNTB0,TCMPB0;stop
        local_irq_restore(flag);
        free_irq(IRQ_TIMER0, NULL);
    }
    return(0);
}
```

4. 2. 4 pwm_ioctl

完成 3 个功能：使能、禁止、读取 PWM 输出的波形数量，源程序：

```
static int pwm_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long
arg){
     switch(cmd){
          case 0:
                 local_irq_save(flag);
                 RunTime = 0;
                 TCON = (TCON & 0x7FFFE0) | 0x09;//Timer0:auto reload; Update TCNTB0, TCMPB0;
             start
                 local_irq_restore(flag);
                 break;
          case 1:
                 local_irq_save(flag);
                 TCON = (TCON & 0x7FFFE0) | 0x0A;//Timer0:auto reload; Update TCNTB0, TCMPB0;
             stop
                 local_irq_restore(flag);
                 break;
                 case 2:
                 if (put_user(RunTime, (u32 *)arg)){
                    return -EFAULT;
                 }
                 break;
          case 3:
                 RunTime = 0;
                 if (put_user(RunTime, (u32 *)arg)){
                    return -EFAULT;
                 }
```

```
            break;
        default:
            return -ENOTTY;
            break;
    }
    return 0;
}
```

4.2.5 pwm_irq_handle

定时中断函数，用于递增PWM输出的波形数量值，源程序：

```
static void pwm_irq_handle(int irq, void *dev_id, struct pt_regs *regs){
    RunTime++;
}
```

4.3 PWM测试程序

主要测试能否正常使能、禁止PWM输出，及频率、占空比是否正确。

```
int main(void){
    int fd;
    unsigned int RunTime;
    unsigned char i;
    fd = open("/dev/PWM", 0);
    if (fd < 0){
        perror("open device PWM");
        exit(1);
    }
    while (1){
        ioctl(fd,3,&RunTime);
        for(i = 0; i < 10; i++){
            ioctl(fd,0,&RunTime);                //启动定时器
            sleep(1);
            ioctl(fd,2,&RunTime);                //读取RunTime值
            printf("1s:RunTime = %d\n",RunTime);
            ioctl(fd,1,&RunTime);                //停止定时器
            sleep(1);
            ioctl(fd,2,&RunTime);                //读取RunTime值
            printf("1s:RunTime = %d\n",RunTime);
        }
        for (i = 0; i < 10; i++){
            ioctl(fd,0,&RunTime);
            sleep(2);
            ioctl(fd,2,&RunTime);
            printf("2s:RunTime = %d\n",RunTime);
            ioctl(fd,1,&RunTime);
            sleep(2);
```

```
            ioctl(fd,2,&RunTime);                //读取 RunTime 值
            printf("2s:RunTime = %d\n",RunTime);
        }
    }
    close(fd);
    return 0;
}
```

5．点亮目标板 LCD

5．1 修改现有的 s3c2410fb.c 驱动程序

Linuette(mizi linux，下同)内核源码树下原有的 s3c2410fb.c(kernel/drivers/video/s3c2410fb.c)是 240*480LCD 的驱动程序，所以要将其修改成自己板子的 640 * 480，其中部份参数需要根据特定的屏作些调整，修改部分如下：

1> 将__initdata 数据结构修改成如下所示：

```
static struct s3c2410fb_mach_info xxx_stn_info __initdata = {
     pixclock:174757,       bpp:     16,
#ifdef CONFIG_FB_S3C2410_EMUL
xres:     96,
#else
xres:     640,
#endif
yres:     480,
hsync_len : 96, vsync_len : 2,
left_margin : 40, upper_margin : 24,
right_margin: 32, lower_margin : 11,

sync:0,cmap_static:1,
reg : {
    lcdcon1 : LCD1_BPP_16T | LCD1_PNR_TFT | LCD1_CLKVAL(1) ,
    lcdcon2 : LCD2_VBPD(32) | LCD2_VFPD(9) | LCD2_VSPW(1),
    lcdcon3 : LCD3_HBPD(47) | LCD3_HFPD(15),
    lcdcon4 : LCD4_HSPW(95) | LCD4_MVAL(13),
    lcdcon5 : LCD5_FRM565 | LCD5_INVVLINE | LCD5_INVVFRAME | LCD5_HWSWP | LCD5_PWREN,
    },
};
```

2> 查找源文件中的 LCDLPCSEL，修改成 LCDLPCSEL &= (~7)；

如果修改成功，内核启动时，左上角会出现 mizilinux 字符和小蜻蜓的 LOGO。也可以将它的单个文件进行编译以模块形式装载测试，装载前先删除原来的文件：

```
#rm /dev/fb/1
#insmod ./s3c2410fb.o
```

自动生成设备文件/dev/fb/1

5．2 LCD 测试程序

```
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/mman.h>
#include <linux/fb.h>
#define FBDEV "/dev/fb/1"
static char *default_framebuffer=FBDEV;
struct fb_dev{
    int fb;
    void *fb_mem;
    int fb_width, fb_height, fb_line_len, fb_size;
    int fb_bpp;
};
static struct fb_dev fbdev;
static void draw(int color){
    int i, j;
    unsigned short int *p = (unsigned short int *)fbdev.fb_mem;
    for (i = 0; i < fbdev.fb_height; i++, p += fbdev.fb_line_len/2){
        for (j = 0; j < fbdev.fb_width; j++){
            p[j] = color;
        }
    }
}
int framebuffer_open(void){
    int fb;
    struct fb_var_screeninfo fb_vinfo;
    struct fb_fix_screeninfo fb_finfo;
    char *fb_dev_name = NULL;
    if (!(fb_dev_name = getenv("FRAMEBUFFER"))){
        fb_dev_name = default_framebuffer;
    }
    fb = open(fb_dev_name, O_RDWR);
    if (fb < 0){
        printf("device %s open failed\n", fb_dev_name);
        return -1;
    }
    if (ioctl(fb, FBIOGET_VSCREENINFO, &fb_vinfo)){
        printf("Can't get VSCREENINFO: %s\n", strerror(errno));
        close(fb);
        return -1;
    }
```

```
        if (ioctl(fb, FBIOGET_FSCREENINFO, &fb_finfo)) {
            printf("Can't get FSCREENINFO: %s\n", strerror(errno));
            return 1;
        }
        fbdev.fb_bpp = fb_vinfo.red.length + fb_vinfo.green.length + fb_vinfo.blue.length +
fb_vinfo.transp.length;
        fbdev.fb_width = fb_vinfo.xres;
        fbdev.fb_height = fb_vinfo.yres;
        fbdev.fb_line_len = fb_finfo.line_length;
        fbdev.fb_size = fb_finfo.smem_len;
        printf("frame buffer: %d(%d)x%d, %dbpp 0x%xbyte\n", fbdev.fb_width, fbdev.fb_line_len,
fbdev.fb_height, fbdev.fb_bpp, fbdev.fb_size);
        if (fbdev.fb_bpp != 16){
            printf("frame buffer must be 16bpp mode\n");
            exit(0);
        }
        fbdev.fb_mem = mmap(NULL, fbdev.fb_size, PROT_READ|PROT_WRITE, MAP_SHARED, fb, 0);
        if (fbdev.fb_mem == NULL || (int)fbdev.fb_mem == -1){
            fbdev.fb_mem = NULL;
            printf("mmap failed\n");
            close(fb);
            return -1;
        }
        fbdev.fb = fb;
        memset(fbdev.fb_mem, 0x0, fbdev.fb_size);
        return 0;
    }
    void framebuffer_close(){
        if (fbdev.fb_mem){
            munmap(fbdev.fb_mem, fbdev.fb_size);
            fbdev.fb_mem = NULL;
        }
        if (fbdev.fb){
            close(fbdev.fb);
            fbdev.fb = 0;
        }
    }
    int main(void){
        int i;
        framebuffer_open();
        for (i = 0; i < 16; i++){
            printf("%d:color = 0x%x", i, 1 << i);
            draw(1 << i);
            getchar();
```

```
    }
    framebuffer_close();
    return 0;
}
```

6．安装触摸板

6．1 触摸板驱动程序的修改

内核自带的触摸板驱动源文件： kernel\drivers\char\s3c2410-ts.c，需在 s3c2410_ts_init()初始化函数中加入“ADCDLY = 0xFFFF”语句，否则在按下触屏时，触屏的采集值将非常的不稳定，误差很大。

6．2 触摸板测试程序

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
typedef struct{
    unsigned short pressure;
    unsigned short x;
    unsigned short y;
    unsigned short pad;
}TS_EVENT;
int main(void){
    static int ts = -1;
    static int mousex = 0;
    static int mousey = 0;
    static TS_EVENT ts_event;
    printf("touch screen test program\n");
    printf("please touch the screen\n");
    ts = open("/dev/touchscreen/0raw",O_RDONLY);
    if (ts < 0) {
        fprintf(stderr,"Can not open touch screen! \n");
        exit(1);
    }
    while (1){
        read(ts, &ts_event, sizeof(TS_EVENT));
        if (ts_event.pressure > 0){
            printf(ts_event.x = %d, ts_event.y = %d\n", ts_event.x, ts_event.y);
        }
    }
}
```

三．构建完整的嵌入式 Linux 系统

1．桌面系统的启动

当触摸板和 LCD 都能正常工作时，就可以启动 Linuette 的桌面系统，将原来的 root.cramfs 根文件系统，更改为带 GUI 的根文件系统（如 root_english.cramfs），内核使用更新过的。

进入 vivi 命令行模式，重新分区：0 192k 1M

下载：vivi zImage root.cramfs

如果系统已正常启动，则省略上述步骤。

启动系统后，如果一切正常将在左上角会出现蜻蜓和 mizilinux 的 logo。

进入系统后，通过 ztelnet 将 imagewrite、root_english.cramfs 下载到目标系统的/tmp 目录，再利用 imagewrite 将文件系统更新成 root_english.cramfs。

#./imagewrite /dev/mtd/0 root_english.cramfs:1M

注：必须先通过 vivi 下载 root.cramfs，待系统正常启动后，通过 ztelnet 下载 root_english.cramfs 和 imagewrite，再使用 imagewrite 烧写 root_english.cramfs 根文件系统，如果直接用 vivi 的 xmodem 下载 root_english.cramfs 需很长的时间，且会下载出错。

重新启动系统将进入触摸屏校准程序，校准正常后进入 QT 桌面环境。在控制台（超级终端）下可以按“Ctrl + C”退出 QT 服务器，而进入平时的命令行模式。/usr/qt/bin 目录下的 cal.sh、start.sh 分别为校准和 QT 服务器启动脚本，都可以重新运行。

2．下载可读写的文件系统 Yaffs

2．1 安装

注：必须用 vivi 进行分区，再用 vivi 下载内核和根文件系统，而不能启动目标板的 linux 系统后，用 imagewrite 写入，否则下载完重启时，系统将不能正常引导。另外 mizi 公司没有在该光盘中提供 yaffs 文件系统的源代码，只提供了编译好的测试内核(zImage.yaffs)。所以上述的内核中也就不支持该文件系统，必须使用编译好的 zImage.yaffs 内核，在 linuette/target/box/image/rwimage 目录下。

vivi>bon part 0 192k 1M 3M:M //注：最后的 M 不能省，它表示 MTD 分区（不带 M 的为 bon 分区），否则下载启动后将不出现 yaffs 文件系统相关的块驱动（/dev/mtd/1 和/dev/mtdblock/1），用 df 显示文件系统时，也不会包括/dev/mtdblock/1 指定的文件系统项

vivi>load flash vivi x

vivi>load flash kernel x　　　//写入 zImage.yaffs

vivi>load flash root x　　　　//写入 root.cramfs

注：为 rwimage 目录下的 root.cramfs，它与原先 root.cramfs 不同的是在启动时会自动挂载 yaffs

vivi>boot

启动系统后输入“#df”命令，会查看到比原来安装的内核多出“/dev/mtdblock/1”指定的文件系统目录“/usr”(说明“mount －t yaffs /dev/mtdblock/1 /usr”成功，yaffs 有效)，此时就可以在/usr 目录下创建目录和保存文件，掉电后不会丢失。如下：

#mkdir /usr/ccn

重启系统

#ls /usr

2．2　yaffs 的制作

2．2．1 生成根文件系统

解压/linuette/target/box/root_dir/rwimage 下的 root.tar 文件

#tar jxvf root.tar.bz2

#./mkcramfs ./root ./root.cramfs

rwimage 目录下的该 root.cramfs 与以前的 root.cramfs 最大区别是在启动过程中会自动挂载 yaffs。

2．2．2 生成 yaffs 类型的 usr 系统

#tar jxvf usr.tar.bz2

#./mkyaffsimage ./usr usr.yaffs c

注：mkyaffsimage 的第四个参数不能等于 convert，这与大小端有关，可以查看 mkyaffsimage 的源代码。

2．3 烧写 yaffs 类型的文件系统到 /usr 目录

除了上述直接在/usr 目录下创建使用 yaffs 系统外，也可以将整个 yaffs 类型的系统(usr.yaffs)烧写到块设备（/dev/mtdblock/1）下，然后挂载到/usr 目录下，也可以在根文件系统的/tmp 目录下再创建一个目录(如：/tmp/yaffs)，然后将/dev/mtdblock/1 挂载到该目录下。下述为挂载到/usr 目录下的方法：

通过 ztelnet 或其它方法将 mkyaffs 和 usr.yaffs 两个文件下载到目标板上执行下述命令

#umount /usr

#./mkyaffs -e /dev/mtd/1 usr.yaffs 或#./mkyaffs /dev/mtd/1 usr.yaffs

将打印出写 NAND Flash 的许多信息，写完后打印出 OK

#mount -t yaffs /dev/mtdblock/1 /usr

此时用#ls /usr 命令可以显示该目录下的文件夹，也可以新键一个目录，再写入文件进行读写测试，如下：

```
#df -h
Filesystem        Size    Used    Available    Use     Mounted on
/dev/bon/2        2.8M    2.8M    0            100%    /
tmpfs             0.6M    0       30.6M        0%      /dev/shm
/dev/mtdblock/1   61.0M   25.3M   35.7M        41%     /usr
#cd /usr; mkdir ccn
#cp /tmp/mkyaffs ./ccn
#cp /tmp/usr.yaffs ./ccn
# ls
mkyaffs    usr.yaffs
#df -h
Filesystem        Size    Used    Available    Use     Mounted on
/dev/bon/2        2.8M    2.8M    0            100%    /
tmpfs             0.6M    0       30.6M        0%      /dev/shm
/dev/mtdblock/1   61.0M   51.3M   9.7M         84%     /usr
```

最后重新启动系统，测试新建的 ccn 目录及内容是否还存在。另外由于 usr.yaffs 里包含了 GUI，所以和 root_english.cramfs 作为根文件系统一样，在启动 console 前会先运行触屏校准程序，再启动 QT 服务器，进入桌面系统。可按 ctrl + C 停止 QT 服务器的运行，而进入 console 模式下查看先前的目录。

3．Yaffs 文件系统移植

当使用 linuette 现成提供的 zImage.yaffs 和 usr.yaffs 是不能正常启动桌面系统的，因为它们提供的包含 yaffs 的测试内核，支持的 LCD 是 320*480，以及触摸板的驱动也没有修改，所以必须在原来的内核源代码中移植 yaffs 文件系统，才能在支持 yaffs 系统的同时，任意修改内核代码部分，如 LCD 分辨率、触摸板，以及将驱动编译进内核等。

移植调试思路：参照 mizi 提供的 yaffs（源码不公开）的使用，及输出信息为线索，一步步的使移植的功能接近它。如：1> 启动时的分区信息（BON info、MTD partitions）2> /dev/mtd 和/dev/mtdblock 3> /proc/filesystems 4> mount －t yaffs /dev/mtdblock/1 /usr

3．1 在 vivi 中进行分区

```
vivi>bon part 0 192k 1M 3M:M
```

注：最后面的 M 表示为 mtd 分区，下面的 flag 将为 00000001；当然也可以只分 3 个区。

```
viiv>bon part show
BON info. (4 partitions)
No: offset     size          flags     bad
---------------------------------------------
 0: 0x00000000 0x00030000    00000000    0   192k
 1: 0x00030000 0x000d0000    00000000    0   832k
 2: 0x00100000 0x00200000    00000000    0     2M
 3: 0x00300000 0x03cfc000    00000001    0   60M+1008k
vivi>load flash vivi x
vivi>load flash kernel x
vivi>load flash root x
vivi>boot
……
NAND device: Manufacturer ID: 0xec, Chip ID: 0x76 (Samsung NAND 64MiB 3,3V)

bon0: 00000000-00030000 (00030000) 00000000
bon1: 00030000-00100000 (000d0000) 00000000
bon2: 00100000-00300000 (00200000) 00000000
bon3: 00300000-03ffc000 (03cfc000) 00000001
……
```

注意：也可以只分 3 个区：

```
vivi> bon part 0 192k 1M
```

3．2 修改 MTD 驱动的 NAND 分区

修改 kernel/drivers/mtd/nand/smc_s3c2410.c 文件中的 smc_partitions[]，如下：

```
static struct mtd_partition smc_partitions[] = {
 {
     name:         "bon",
```

```
        size:           0x04000000,
        offset:         0x0,
        mask_flags:     MTD_WRITEABLE,      /*force read-only */
    }, {
        name:           "mtd",
        size:           0x03cfc000,
        offset:         0x00300000,
    }
};
```

另外在该文件开始部分的消除分区宏定义：

```
#undef CONFIG_MTD_SMC_S3C2410_SMDK_PARTITION
```

修改成：

```
#define CONFIG_MTD_SMC_S3C2410_SMDK_PARTITION
```

内核配置时选中Memory Technology Devices (MTD) ---> NAND Flash Device Drivers ---> SMC Device Support 及下属所有项

重新编译内核，下载，启动。输出信息如下：

```
……
NAND device: Manufacture ID: 0xec, Chip ID: 0x76 (Samsung K9D1208V0M)
Creating 2 MTD partitions on "Samsung K9D1208V0M":
0x00000000-0x04000000 : "bon"
0x00300000-0x03ffc000 : "mtd"
bon0: 00000000-00030000 (00030000) 00000000
bon1: 00030000-00100000 (000d0000) 00000000
bon2: 00100000-00300000 (00200000) 00000000
bon3: 00300000-03ffc000 (03cfc000) 00000001
……
#cat /proc/mtd
dev:    size   erasesize  name
mtd0: 04000000 00004000 "bon"
mtd1: 03cfc000 00004000 "mtd"
#ls /dev/mtd
0     0ro   1     1ro
#ls /dev/mtdblock
0   1
```

3．3 移植 yaffs 源代码

从官网（http://www.aleph1.co.uk/cgi-bin/viewcvs.cgi/yaffs/）下载源码（yaffs.tar.gz）解压，在kernel/fs/目录下新建yaffs目录，将源码（yaffs_fs.c、yaffs_guts.c、yaffs_mtdif.c、yaffs_ecc.c、devextras.h、yaffs_guts.h、yaffs_mtdif.h、yaffs_ecc.h、yaffsinterface.h、yportenv.h）复制到该目录下，创建yaffs_config.h文件，加入源码中使用到的宏定义，如下：

```
#ifndef __YAFFS_CONFIG_H__
#define __YAFFS_CONFIG_H__
#define CONFIG_YAFFS_MTD_ENABLED
```

#define CONFIG_YAFFS_USE_OLD_MTD

#endif

在该目录下创建 Makefile 文件，内容如下：

```
O_TARGET := yaffs.o

obj-y := yaffs_fs.o yaffs_guts.o yaffs_mtdif.o yaffs_ecc.o
obj-m := $(O_TAGRET)

include $(TOPDIR)/Rules.make
```

在 fs 目录的 Makefile 文件中加入 yaffs 子目录，如下：

subdir-$(CONFIG_YAFFS_FS) += yaffs

在 fs 目录的 Config.in 文件中加入 YAFFS 选项，如下：

if [“$CONFIG_MTD_SMC” = “y”]; then

tristate ‘Yaffs filesystem on NAND’ CONFIG_YAFFS_FS

fi

注：if 的[ ]与字符间要用空隔隔开，否则 if 条件会始终不成立

内核配置时选中 File system ---> Yaffs filesystem on SMC，再重新编译，下载，启动。输出信息如下：

```
#cat /proc/filesystems
nodev rootfs
nodev bdev
nodev proc
nodev sockfs
nodev tmpfs
nodev shm
nodev pipefs
cramfs
nodev ramfs
vfat
nodev devfs
nodev nfs
jffs2
nodev autofs
nodev devpts
yaffs
#mount -t yaffs /dev/mtdblock/1 /usr
yaffs: dev is 7937 name is "1f:01"
#mkdir /usr/ccn
#ls /usr
ccn          lost+found
#df -h
Filesystem                Size      Used Available UseMounted on
/dev/bon/2                2.7M      2.7M         0 100/
tmpfs                    30.6M         0     30.6M   0/dev/shm
```

/dev/mtdblock/1 61.0M 100.0k 60.9M 0/usr

3．4 出现的问题

3．4．1 在 smc_partitions[]中添加 MTD 分区不起作用，如下：

……

NAND device: Manufacture ID: 0xec, Chip ID: 0x76 (Samsung K9D1208V0M)

bon0: 00000000-00030000 (00030000) 00000000

bon1: 00030000-00100000 (000d0000) 00000000

bon2: 00100000-00300000 (00200000) 00000000

bon3: 00300000-03ffc000 (03cfc000) 00000001

……

#ls /dev/mtd

0 0ro

#ls /dev/mtdblock

0

#cat /proc/mtd

dev: size erasesize name

mtd0: 04000000 00004000 "Samsung K9D1208V0M"

原因是定义分区的 smc_partitions[]在 CONFIG_MTD_SMC_S3C2410_SMDK_PARTITION 宏定义内，而在该文件的开始又消除该宏定义，因此必须将消除修改成定义，如：

#undef CONFIG_MTD_SMC_S3C2410_SMDK_PARTITION

修改成：

#define CONFIG_MTD_SMC_S3C2410_SMDK_PARTITION

3．4．2 在 fs 目录的 Config.in 文件中加入 YAFFS 选项，但在内核配置中始终没有出现 File system ---> Yaffs filesystem on SMC 项。

原因：Config.in 中 if 语句的[ ]与字符间要用空隔隔开，否则 if 条件会始终不成立，如下：

if [“$CONFIG_MTD_SMC” = “y”]; then

tristate ‘Yaffs filesystem on NAND’ CONFIG_YAFFS_FS

fi

3．4．3 将加入 yaffs 的内核下载目标板启动后，仍没有 yaffs 项，如下：

cat /proc/filesystems

nodev rootfs

nodev bdev

nodev proc

nodev sockfs

nodev tmpfs

nodev shm

nodev pipefs

cramfs

nodev ramfs

vfat

nodev devfs

nodev nfs

```
nodev autofs
nodev devpts
#
```

在 yaffs_fs.c 文件的 init_yaffs_fs()函数的开始和注册文件系统语句前增加打印信息，如下：

```
static int __init init_yaffs_fs(void){
    int error = 0;
    yaffs_dev = yaffsram_dev = NULL;
    printk(“init_yaffs_fs…….\n”);
    printk(KERN_DEBUG "yaffs " __DATE__ " " __TIME__ " Initialisation\n");
    ......
#ifdef CONFIG_YAFFS_MTD_ENABLED
    printk(“register yaffs filesystem………”);
    error = register_filesystem(&yaffs_fs_type);
    if (error){
#ifdef CONFIG_YAFFS_RAM_ENABLED
        unregister_filesystem(&yaffs_ram_fs_type);
#endif //CONFIG_YAFFS_RAM_ENABLED
    }
#endif // CONFIG_YAFFS_MTD_ENABLED
    return error;
}
```

重新编译，下载，启动信息如下：

```
……
devfs: v1.10 (20020120) Richard Gooch (rgooch@atnf.csiro.au)
devfs: boot_options: 0x1
init_yaffs_fs...............
ttyS00 at I/O 0x50000000 (irq = 52) is a S3C2410
ttyS01 at I/O 0x50004000 (irq = 55) is a S3C2410
……
```

由上述信息中只有“init_yaffs_fs…….”而没有“register yaffs filesystem……”，说明系统调用了 yaffs 初始化函数，但没有执行文件系统注册函数，所以在/proc/filesystems 中也就没有 yaffs 项。必须增加 CONFIG_YAFFS_MTD_ENABLED 的宏定义，才能使该部分的注册函数有效，故在文件开始前增加：

```
#defined CONFIG_YAFFS_MTD_ENABLED
```

或者在源码目录下创建 yaffs_config.h 文件，定义源码中所有的宏定义，再被其它源文件包含。

再次编译，下载，启动信息如下：

```
……
devfs: v1.10 (20020120) Richard Gooch (rgooch@atnf.csiro.au)
devfs: boot_options: 0x1
init_yaffs_fs...............
register_filesystem...................
ttyS00 at I/O 0x50000000 (irq = 52) is a S3C2410
ttyS01 at I/O 0x50004000 (irq = 55) is a S3C2410
……
#cat /proc/filesystems
```

nodev rootfs

nodev bdev

nodev proc

nodev sockfs

nodev tmpfs

nodev shm

nodev pipefs

cramfs

nodev ramfs

vfat

nodev devfs

nodev nfs

nodev autofs

nodev devpts

yaffs

#

3．4．4 包含 CONFIG_YAFFS_MTD_ENABLED 宏定义后，编译出现错误：

yaffs_mtdif.c:33 variable ‘yaffs_oobinfo’ has initializer but incomplete type

yaffs_mtdif.c:34 unknow field ‘useecc’ specified in initializer

……

将 yaffs_mtdif.c 中的用 nand_oobinfo 定义的两个结构体屏蔽掉

3．4．5 #mount －t yaffs /dev/mtdblock/1 /usr 出错

当执行该命令时会调用 yaffs_fs.c 文件中的 yaffs_internal_read_super()函数，主要还是该函数及子函数出现错误，可以用 printk 函数打印信息的方法跟踪解决。本错误是由于当时没有将 yaffs_mtdif.c 文件中的 nand_oobinfo 结构完成屏蔽，而只是屏蔽掉了编译时出现错误的 useecc，所以在调试时出现一系列对未初始化指针操作等错误。所以如果仔细按照上述的移植步骤操作，应不会出现类似的错误。

3．4．6 #./mkyaffs －e /dev/mtd/1 usr.yaffs 出现下述错误：

argc 4 sh 0 optcnt 2

MEMSETOOBSEL:Inappropriate ioctl for device

打开 yaffs 源码包 utils 目录下的 mkyaffs.c 文件（mkyaffs 工具的源文件），可以看出是下例语句出现了错误：

```
Oobsel = usemtdecc ? yaffs_oobinfo : yaffs_noeccinfo;
if (ioctl(fd, MEMSETOOBSEL, &oobsel) != 0){
    perror (“MEMSETOOBSEL”);
    close (fd);
    exit (1);
}
```

该功能给/dev/mtd/1 设备传递写入 nand flash 的 ecc 信息（mtdchar.c 文件的 ioctl()函数），和上面屏蔽掉的 yaffs_mtdif.c 文件中的 yaffs_oobinfo 等是同一功能。由于我们使用的 MTD 驱动比较旧，查看/drivers/mtd/mtdchar.c 文件的 ioctl()函数可知还不支持该功能。因此也必须将 mkyaffs.c 文件中的该功能去掉再重新编译生成 mkyaffs。或者使用原来的 mkyaffs 工具，但不加-e 参数，如：

#./mkyaffs /dev/mtd/1 usr.yaffs

或直接修改该源文件去掉-e 功能。

3．4．7 mkyaffs 编译出错

在 Makefile 文件中的 MAKETOOLS = 处添加交叉编译工具路径，如：

MAKETOOLS = /opt/host/armv4l/bin/armv4l-unknown-linux-

将#include <mtd/mtd-user.h>替换成#include <linux/mtd/mtd.h>；将 nand_oobinfo 两个结构屏蔽掉；将 main 函数中的 struct nand_oobinfo oobsel;和问题 6 提到的与 MEMSETOOBSEL 相关的语句屏蔽掉。

最后编译生成目标文件 mkyaffs_noecc（为了与原来的 mkyaffs 区别加上_noecc 后缀）。

3．4．8 系统启动后/proc 目录为空等问题，如下：

#df －h

Filesystem   size   Used Available Use% Mounted on

df: /proc/mounts: No such file or directory

#ls /proc

而且，在宿主机上也不能用 ztelnet 登录，提示：

[root@localhost root_dir]#ztelnet 192.168.1.7

Trying 192.168.1.7…

telnet: Unable to connect to remote host: Connection refused

原因是在 vivi 中的 kernel command line 参数有错误，如下述系统启动时的信息中缺少 linuxrc 脚本：

Kernel command line: console=ttyS0 root=/dev/bon/2

在 vivi 下应设置成：

vivi> param set linux_cmd_line “noinitrd root=/dev/bon/2 init=linuxrc console=ttyS0”

启动后信息如下：

Kernel command line: noinitrd root=/dev/bon/2 init=linuxrc console=ttyS0

另外，yaffs 文件系统在多次使用，没有可用空间时，也可能出现该现象，需在 vivi 中重新分区，再下载内核、文件系统等解决。

3．4．9 挂载 yaffs 文件系统时出现错误，如下：

#mount －t yaffs /dev/mtdblock/1 /usr

yaffs: Attempting MTD mount on 31.1, “if:01”

mout: Mounting /dev/mtdblock/1 on /usr failed: Not a directory

原因是在执行 mkyaffsimage 工具打包 yaffs 格式文件系统影像时，加入了 convert 参数，如

在 PC 机上：

[root@localhost root_dir]#./mkyaffsimage ./root_english ./root_english.yaffs convert

‘convert’ produce a big-endian image from a little-endian machine

在目标板上：

#./mkyaffs /dev/mtd/1 ./root_english.yaffs

……

OK

#mount －t yaffs /dev/mtdblock/1 /usr

yaffs: Attempting MTD mount on 31.1, “if:01”

mout: Mounting /dev/mtdblock/1 on /usr failed: Not a directory

只需将刚才的 MTD 设备重新用 yaffs 系统格式化就可以，如下：

#./mkyaffs /dev/mtd/1

……

Erasing block at 0x83cf8000

OK

#mount －t yaffs /dev/mtdblock/1 /usr

yaffs: Attempting MTD mount on 31.1, “if:01”

#

制作 yaffs 文件系统的镜像不应加 convert 参数，但又要满足参数大于 3 个，所以第四个参数只要不为 convert 就可以，

PC 机如下：

[root@localhost root_dir]#./mkyaffsimage ./root_english ./root_english.yaffs c

下载到目标板后执行：

#./mkyaffs /dev/mtd/1 ./root_english.yaffs

……

OK

#mount –t yaffs /dev/mtdblock/1 /usr

yaffs: Attempting MTD mount on 31.1, “if:01”

#

3．5 yaffs 初始化及执行过程的简单总结

Step 1： 先调用 yaffs_fs.c 文件中的 init_yaffs_fs()初始化函数，它负责 yaffs 文件系统的注册，将 yaffs 添加到/proc/filesystems 中；启动信息：

……

devfs: v1.10 (20020120) Richard Gooch (rgooch@atnf.csiro.au)

devfs: boot_options: 0x1

init_yaffs_fs.............. （需要在函数中加该打印信息）

register_filesystem..................

……

Step 2： 调用 drivers/mtd/nand/smc_s3c2410.c 文件的 smc_s3c2410_init()函数扫描及初始化 NAND Flash 设备及底层接口函数，和 mtd 分区；启动信息：

……

NAND device: Manufacture ID: 0xec, Chip ID: 0x76 (Samsung K9D1208V0M)

add_mtd_partitions... （需要在函数中加该打印信息）

Creating 2 MTD partitions on "Samsung K9D1208V0M":

0x00000000-0x04000000 : "bon"

0x00300000-0x03ffc000 : "mtd"

……

Step 3 ： 执行 mount –t yaffs /dev/mtdblock/1 /usr 命令时，调用 yaffs_fs.c 文件中的 yaffs_internal_read_super()函数。执行信息：

mount -t yaffs /dev/mtdblock/1 /usr

yaffs_internal_read_super................ （需要在函数中加该打印信息）

yaffs: Attempting MTD mount on 31.1, "1f:01"

Step 4： 另外还有对 yaffs 文件系统下的读或写等等，都会调用 yaffs_fs.c 文件下的相应函数，再调用 mtd 设备对应的 NAND Flash 接口函数等。

4. Yaffs 作为根文件系统启动

4. 1 直接复制的方法更新 yaffs —— 将 root 内容复制到 yaffs 文件系统/usr 后启动

```
vivi>bon part 0 192k 1M 3M:M
vivi>load flash vivi x
vivi>load flash kernel x              //支持 yaffs 的内核
vivi>load flash root x
vivi>boot
……
#ifconfig eth0 192.168.1.7
#inetd                                //启动 ztelnet 服务
#mount －t yaffs /dev/mtdblock/1 /usr
```

在 PC 机上进入/linuette/target/box/root_dir，进入根文件系统 root（也可是 root_english，或其它）目录下，再将其所有内容打包

[root@localhost root]#tar zcvf root.tgz *

[root@localhost root]#ztelnet 192.168.1.7

通过 ztelnet 将 root.tgz 下载到目标板的/usr 目录下

在目标板上：

cd /usr

tar zxvf root.tgz;rm －rf root.tgz

解压完后重启进入 vivi，修改启动参数

vivi>param show

……

Linux command line: noinitrd root=/dev/bon/2 init=linuxrc console=ttyS0

vivi>param set linux_cmd_line “noinitrd root=/dev/mtdblock/1 init=linuxrc console=ttyS0”

vivi>param save

……

vivi>boot

此时的根文件系统为可写的 yaffs 文件系统，可以在根目录下创建目录、文件等测试。

4. 2 使用 mkyaffsimage 工具更新 Yaffs —— 将 root_english 制作成 Yaffs 根文件系统

上述是直接将文件系统下的所有文件简单的复制到挂载 yaffs 文件系统的/usr 目录，也可以利用先前提到的 mkyaffsimage 和 mkyaffs_noecc 工具完成。

PC 宿主机下：

[root@localhost /]#cd /linuette/target/box/roo_dir/

[root@localhost root_dir]#./mkyaffsimage ./root_english ./root_english.yaffs c

[root@localhost root_dir]#ztelnet 192.168.1.7

通过 ztelnet 将 mkyaffs_noecc、root_english.yaffs 下载到目标板的/tmp 目录下

假设此时目标板根文件系统为 root.cramfs，且未挂载 yaffs，如果已经将 yaffs 挂载到了/usr，需要先#umount /usr

目标板下：

#./mkyaffs_noecc /dev/mtd/1 ./root_english.yaffs

……

OK

查看 root_english 是否生成成功

#mount －t yaffs /dev/mtdblock/1 /usr

yaffs: Attempting MTD mount on 31.1, “if:01”

#ls /usr

可与宿主机上的/linuette/target/box/root_dir/root_english 目录比较是否有错误

重启动进入 vivi，修改启动参数

vivi>param set linux_cmd_line “noinitrd root=/dev/mtdblock/1 init=linuxrc console=ttyS0”

vivi>param save

vivi>boot

5．完整的嵌入式 Linux 系统

目前有了针对 nand flash 的可读写文件系统 Yaffs，加上先前已正常工作的 LCD、触摸板等内核驱动，我想该平台可以称是比较完整的嵌入式 Linux 系统平台了。先前提到的嵌入式 Linux 软件系统一般有由 bootloader、kernel、root filesysytem 和可选的 user filesystem 构成，则我们可有如下选择：

不带 GUI 时

平台 1：vivi + zImage + root.cramfs

平台 2：vivi + zImage + root.cramfs + user（user 为 yaffs 格式的空目录，不含 GUI）

平台 3：vivi + zImage + root.yaffs

带 GUI 时

平台 4：vivi + zImage + root_english.cramfs（或 root_china.cramfs 等，下同）

平台 5：vivi + zImage + root_english.yaffs

平台 6：vivi + zImage + root.cramfs + usr.yaffs

……

四．嵌入式 WEB 服务器

1．Boa 移植

1．1 Boa 移植步骤

Step 1：从www.boa.org下载Boa源码，将其解压并进入源码目录的src子目录。

#tar xzf boa-0.94.13.tar.gz

#cd boa-0.94.13/src

生成 Makefile 文件

#./configure

修改 Makefile 文件，找到 CC=gcc 和 CPP=gcc -E，分别将其改为交叉编译器安装的路径 CC=/opt/host/armv4l/bin/armv4l-unknown-linux-gcc 和 CPP=/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -E 并保存退出。

然后运行 make 进行编译，得到可执行程序 boa

#make

去掉调试信息

#/opt/host/armv4l/bin/armv4l-unknown-linux-strip boa

Step 2：Boa 的配置。

Boa 需要在/etc 目录下建立一个 boa 目录，里面放入 Boa 的主要配置文件 boa.conf。在 Boa 源码目录下已有一个示例 boa.conf，可以在其基础上进行修改。

Group nogroup 修改成 Group 0，由于在/etc/group 文件中没有 nogroup 组，所以设成 0

另外在/etc/passwd 中有 nobody 用户，所以 User nobody 不用修改。

ScriptAlias /cgi-bin/ /usr/lib/cgi-bin/ 修改成 ScriptAlias /cgi-bin/ /var/www/cgi-bin/

其它默认设置即可。

还需要创建日志文件所在目录/var/log/boa，创建 HTML 文档的主目录/var/www，将静态网页存入该目录下（可以将主机/usr/share/doc/HTML/目录下的 index.html 文件和 img 目录复制到/var/www 目录下），创建 CGI 脚本所在目录/var/www/cgi-bin，将 cgi 的脚本存放在该目录下。另外还要将 mime.types 文件复制/etc 目录下，通常可以从 linux 主机的/etc 目录下直接复制即可。

Step 3：测试

1> 静态 HTML 网页

在目标板上运行 boa 程序，将主机与目标机的 ip 设成同一网段，然后打开任一个浏览器 （linux 或 window 下都可），输入目标板的 ip 地址（http：//10.10.10.2）即可打开/var/www/index.html 网页

2> CGI 脚本测试

boa 源码中有 cgi-bin 测试脚本(boa-0.94.13/examples/cgi-test.cgi)，但由于它不是在我们目标平台下编译的，所以不能将它复制到/var/www/cgi-bin 目录下进行测试。

下面是一个简单的 hello, world!程序，代码如下：

```
#include <stdio.h>
void main(){
printf("Content-type: text/html\n\n");
printf("<html>\n");
printf("<head><title>CGI Output</title></head>\n");
printf("<body>\n");
printf("<hl>Hello, world.</hl>\n");
printf("<body>\n");
printf("</html>\n");
exit(0);
}
```

1.2 Boa 移植时出现的问题

1> 当运行 boa 程序时出现错误，如下：

./boa

[27/Nov/1990:13:22:25 + 0000]boa.c:266.icky Linux kernel bug!:No such file

将 User 0 修改成 User nobody

2> 打开网页时，网页中的图片无法显示

就将存放图片的子目录/var/www/images 修改成/var/www/img

3> 在测试 cgi 脚本时，当在浏览器地址中输入“http//10.10.10.2/cgi-bin/helloworld.cgi”时，

浏览输出下述错误：

502 Bad Gateway

The CGI was not CGI/1.1 compliant

在目标板的调试终端中输出下述错误：

……cgi_header：unable to find LFIF

上述错误是在boa原码中的cgi_header.c文件中的process_cgi_header函数产生，如下：

```
    ……
    buf = req->header_line;
    c = strstr(buf, "\n\r\n");
    if (c == NULL) {
        c = strstr(buf, "\n\n");
        if (c == NULL) {
            log_error_time();
            fputs("cgi_header: unable to find LFLF\n", stderr);  //出错信息
#ifdef FASCIST_LOGGING
            log_error_time();
            fprintf(stderr, "\"%s\"\n", buf);
#endif
            send_r_bad_gateway(req);
            return 0;
        }
    }
    ……
```

我们可以定义FASCIST_LOGGING，使产生该错误时将buf内容打印出来，结果如下：

```
……cgi_header：unable to find LFIF
Content-type: text/html
<html>
<head><title>CGI Output</title></head>
<body>
<hl>Hello, world.</hl>
<body>
</html>
```

原来buf中的内容就是helloworld.c中输出的内容，查看输出结果，再看process_cgi_header函数中的语句：c = strstr(buf, "\n\n")，很明显buf中没有两个连续的换行符"\n\n"，所以是helloworld.c文件中的语句：printf("Content-type: text/html\n\n");错写成了 printf("Content-type: text/html\n");

上述行通过标准输出将字符串" Contenttype:text/plain\n\n" 传送给Web服务器。它是一个MIME头信息,告诉Web服务器随后的输出是以纯ASCII文本的形式。在这个头信息中有两个新行符,这是因为Web服务器需要在实际的文本信息开始之前先看见一个空行。

2．WEB应用开发

在实践的产品开发中，一般都使用java小程序，通常先将控制界面的网页放在目标板WEB服务器指定的目录，目标板同时也存放java小程序，当客户浏览器打开控制网页时，获取java小程序（需安装java

虚拟机），java 小程序再与目标板应用程序的网络服务器建立一个连接，实现相互通信。

由于我还没有时间精力学习 jave，所以下面是基于 CGI 的方法。

2．1 CGIC 库的移植

从CGIC的主站点http://www.boutell.com/cgic/上下载源码，将其解压并进入源码目录。

#tar －zxvf cgic.tar.gz

#cd cgic205

修改 Makefile 文件，找到 CC=gcc，将其改为 CC=/opt/host/armv4l/bin/armv4l-unknown-linux-gcc，找到 AR=ar，将其改为 AR=/opt/host/armv4l/bin/armv4l-unknown-linux-ar，找到 RANLIB=ranlib，将其改为 RANLIB=/opt/host/armv4l/bin/armv4l-unknown-linux-ranlib 。找到 gcc cgictest.o －o cgictest.cgi ${LIBS}，将其改为$(CC) $(CFLAGS) cgictest.o －o cgictest.cgi ${LIBS}，找到 gcc capture.o －o capture ${LIBS}，并保存退出。

然后输入 make 进行编译，可以将生成的 capture 和 cgictest.cgi 文件复制到目标板目录 /var/www/cgi-bin/下，以测试正确性。

2．2 基于 CGIC 库的例程

参考《嵌入式 Linux 系统开发详解——基于 EP93XX 系列 ARM》一书中的 list.html 和 list.c

2．3 例程出现的问题

1> 点击 submit(提交)按钮后，终端输出错误信息：…mkstemp：Permission denied

浏览器弹出错误对话框：此文档中无数据

原因：是在 boa 源代码目录下的 util.c 文件中，用 mkstemp()函数创建一个临时文件时出现权限错误。这是由于在 boa.conf 文件设置了 user：nobody，使其运行 boa 服务器时以 nobody 为用户（可以用 ps 命令查看），所以在创建临时文件是没有足够的权限，可以在 boa.conf 中将运行 boa 的用户为 root 身份(user：root)。

2> 当上述设置 user：root，运行 boa 时，在终端会输出错误信息：boa.c:266.icky Linux kernel bug!:No such file

原因：boa.c 文件中的下述语句出现问题，可以将 boa.c 文件中的该行屏蔽掉。

```
if (setuid(0) != -1) {
            DIE("icky Linux kernel bug!");
        }
```

3> 点击例程中的 submit(提交)按钮后浏览器出现下述错误：

地址变为：http://10.10.10.2/var/www/cgi-bin/list.cgi

内容：400 Bad Request Your client has issued a malfarned or illegal request

原因是 list.html 中<form name="SystemCont" method="post" action="/cgi-bin/list.cgi" onSubmit="return checkform()"> 写成了 <form name="SystemCont" method="post" action="/var/www/cgi-bin/list.cgi" onSubmit="return checkform()">

4> 在查看目录项输入目录如：/var/www/cgi-bin/，然后点击 submit(提交)按钮后，浏览器恢复原样，而没有在文本框中显示 ls /var/www/cgi-bin 的内容。

原因是在 list.html 文件中的<!--Dir-->和<!--Files-->的前面有空格，使 list.c 文件中下述语句出

错，所以在list.html中的上面两个注释一定要写在行的开头。

```
if (strcmp(buf,"<!--Dir-->")==0)
{……}
if (strcmp(buf,"<!--Files-->")==0)
{……}
```

2．4 CGI的简单调试

CGI程序像其他可执行程序一样,可通过标准输入(stdin)从Web服务器得到输入信息,如Form中的数据,这就是所谓的向CGI程序传递数据的POST方法。这意味着在操作系统命令行状态可执行CGI程序,对CGI程序进行调试。

CGI程序通过标准输出(stdout)将输出信息传送给Web服务器。传送给Web服务器的信息可以用各种格式,通常是以纯文本或者HTML文本的形式,这样我们就可以在命令行状态调试CGI程序,并且得到它们的输出。

实验下例程序，直接在命令行下调试，主要测试环境变量

```
#include <stdio.h>
#include <stdib.h>
main(){
    int,i,n;
    printf ("Contenttype:text/plain\n\n");
    n=0;
    if(getenv("CONTENT-LENGTH"))
    n=atoi(getenv(CONTENT-LENGTH"));
    for (i=0;i<n;i++)
    putchar(getchar());
    putchar ('\n');
    fflush(stdout);
}
```

五．NFS的配置

尽管ztelnet也可以将主机上的文件传送到目标板上，但还是没有利用NFS将主机的目录直接挂载到目标机的目录下方便。

1．主机的NFS服务器配置

setp 1：编辑共享目录配置文件exports，指定共享目录及权限等。

#vi /etc/exports

在该文件中添加如下内容：

/image 10.10.10.* (rw，sync，no_root_squash)

表示允许IP地址为10.10.10.*的目标机以读写的权限来访问/image目录，也可简单的写成：

/image 10.10.10.* (rw)

setp 2：关闭防火墙(如果不执行这一步，setp 4的回环测试也能通过，但不能被客户端访问)

在文本模式下# setup进行进入设置窗口，在Firewall configration的Security Level项中，选中

No firewall，然后退出。

setp 3：启动 NFS 服务

#/etc/init.d/nfs restart

setp 4：NFS 服务器的回环测试，以验证共享目录能否被访问

#mount 10.10.10.1:/image /mnt/nfs

将共享目录挂载到本机的/mnt/nfs 目录下，如果成功，则 NFS 正常工作。

2. 目标机的 NFS 客户端配置

setp 1：让目标系统的 linux 内核支持 NFS 客户端

在进行内核配置(# make menuconfig)时选中如下项：

File systems -> Network File Systems -> Provide NFSv3 client support

Setp 2：在目标机的 linux 系统目录下挂载 NFS 服务器的共享目录

#cd /tmp

#mkdir nfs

#mount －o nolock 10.10.10.1:/image ./nfs

#ls ./nfs

3. 出现的问题

在目标机上执行#mount －o nolock 10.10.10.1:/image ./nfs 时，总不能成功

原因是没有关闭主机的防火墙，执行#setup 将 Firewall config 的 Security 设置成 NO Firewall，再执行#/etc/init.d/nfs restart，最后再在目标机上#mount －o nolock 10.10.10.1:/image ./nfs

第三阶段 在项目中应用 Linux

公司没有其它的 Linux 工程师，一直以来也没有招到合适的 Linux 工程师，因此领导觉得用 Linux 的技术难度太大，但更不愿意花高价钱引入 VxWorks。可是产品要突破瓶颈，要与对手竞争，早晚而且必须要走这一步。在对产品实现所需的相关技术深入学习评估后，坚定的向领导承诺能够独自实现、解决所有的方方面面。虽然有些害怕、忧虑和信心不足，但却给自己断了所有的退路。事实证明那是是完全正确的，很多东西并没有当初想象的那么复杂，时间也足够。由于产品没有图形界面，不需 GUI，这也省心了不少。原理图参考 mizi 光盘里的，那是比较完整的，再参照对比其它的开发板和书籍，分析它们不同的地方，再扩展产品特有部分。像以太网等很多外围器件的驱动，系统都已自带了，或有些只需修改一小部分，而先增的外围驱动参考先前的《linux 设备驱动程序》一书，难度也不是很大。应用编程，由于对自己产品的熟悉，和参考《GNU/Linux 编程指南》和《UNIX 环境高级编程》，所以只是工作量大些，也不存在难度问题。网络通信可以参考《UNIX 网络编程》，虽然它很厚，但只需参考学习一部分，比想象中要简单很多。WEB 控制肯定不能自己用 CGI 写，找个 PC 机程序员，写个 java 小程序，自己把 boa 服务器放进去，也就是普通的网络编程。WiFi 花的精力比较多，领导也最重视这个功能，而当时对它也最为模糊。我觉得要想在产品上应用它，自己应该先学会它的使用和配置，于是就上网收索，学习在 windows 下配置使用，然后在 linux 下配置使用。因为一般的网卡在 linux 下都没有驱动，又用 ndiswrapper 在 RedHat 9 下驱动无线网卡。由于扩展的 PCMCIA 卡控制器 PD6710 是 16 位的，所以选择了华硕的 WL-110，它没有现成的 linux 驱动，使用的是 linux-wlan 项目的 linux-wlan-ng 驱动程序。也是先在 PC 机上使用 linux-wlan，成功后再在目标板上使用 linux-wlan。

注：以下只记录了我觉得不会对公司利益产生影响的部分内容。

一． 进程间隔定时器

1．概念

所谓“间隔定时器（Interval Timer，简称 itimer）就是指定时器采用“间隔”值（interval）来作为计时方式，当定时器启动后，间隔值 interval 将不断减小。当 interval 值减到 0 时，我们就说该间隔定时器到期。它主要被应用在用户进程上，每个 Linux 进程都有三个相互关联的间隔定时器：

1> 真实间隔定时器（ITIMER_REAL）：这种间隔定时器在启动后，不管进程是否运行，每个时钟滴答都将其间隔计数器减 1。当减到 0 值时，内核向进程发送 SIGALRM 信号。

2> 虚拟间隔定时器 ITIMER_VIRT：也称为进程的用户态间隔定时器。当虚拟间隔定时器启动后，只有当进程在用户态下运行时，一次时钟滴答才能使间隔计数器当前值 it_virt_value 减 1。当减到 0 值时，内核向进程发送 SIGVTALRM 信号（虚拟闹钟信号），并将 it_virt_value 重置为初值 it_virt_incr。

3> PROF 间隔定时器 ITIMER_PROF：当一个进程的 PROF 间隔定时器启动后，则只要该进程处于运行中，而不管是在用户态或核心态下执行，每个时钟滴答都使间隔计数器 it_prof_value 值减 1。当减到 0 值时，内核向进程发送 SIGPROF 信号，并将 it_prof_value 重置为初值 it_prof_incr。

定时器在初始化时，被赋予一个初始值，随时间递减至 0 后发出信号，同时恢复初始值。在任务中我们可以使用一种或全部三种定时器，但同一时刻同一类型的定时器只能使用一个。

2．数据结构

Linux 在 include/linux/time.h 头文件中为上述三种进程间隔定时器定义了索引标识，如下所示：

```
#define ITIMER_REAL 0
#define ITIMER_VIRTUAL 1
#define ITIMER_PROF 2
```

虽然，在内核中间隔定时器的间隔计数器是以时钟滴答次数为单位，但是让用户以时钟滴答为单位来指定间隔定时器的间隔计数器的初值显然是不太方便的，因为用户习惯的时间单位是秒、毫秒或微秒等。所以 Linux 定义了数据结构 itimerval 来让用户以秒或微秒为单位指定间隔定时器的时间间隔值。其定义如下（include/linux/time.h）：

```
struct itimerval {
    struct timeval it_interval;   /* timer interval */
    struct timeval it_value;      /* current value */
};
```

其中，it_interval 成员表示间隔计数器的初始值，而 it_value 成员表示间隔计数器的当前值，它们都是 timeval 结构类型的变量。

```
struct timeval {
    time_t tv_sec;                /* seconds */
    suseconds_t tv_usec;          /* microseconds */
};
```

3. 操作函数

int getitimer(int which, struct itimerval *value);

getitimer()函数得到间隔计时器的时间值并保存在 value 中。有两个参数： 1> which，指定查询调用进程的哪一个间隔定时器，其取值可以是 ITIMER_REAL、ITIMER_VIRT 和 ITIMER_PROF 三者之一。2> value 指针，指向用户空间中的一个 itimerval 结构，用于接收查询结果。

setitimer(int which, struct itimerval *value, struct itimerval *ovalue);

setitimer()不仅设置调用进程的指定间隔定时器，而且还返回该间隔定时器的原有信息。它有三个参数： 1> which，含义与 sys_getitimer()中的参数相同。2> 输入参数 value，指向用户空间中的一个 itimerval 结构，含有待设置的新值。3> 输出参数 ovalue，指向用户空间中的一个 itimerval 结构，用于接收间隔定时器的原有信息。

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

该函数为 signum 所指定的信号设置信号处理器。sigaction 结构(act 和 oldact)描述了信号的部署。它在<signal.h>中的完整定义为：

```
struct sigaction{
 void (*sa_handler)(int);    //规定了当 signum 中的信号产生后，要调用的处理函数
 sigset_t sa_mask;           //定义了在执行处理函数期间应该阻塞的其它信号集合的信号掩码
 int sa_flags;
 void (*sa_restorer)(void);
};
```

4. 测试程序

```
#include <stdio.h>
#include <signal.h>
#include <sys/time.h>
#include <asm/param.h>       //#define HZ
struct timeval tpstart,tpend;
float timeuse;
/* signal process */
static timer_count = 0;
void prompt_info(int signo){
    if ((++timer_count)%100 == 0){
    time_t t=time(NULL);                        //获取秒数表示的系统当前时间
    printf("[%d]prompt_info called", timer_count);
    printf("      current time %s",ctime(&t)); //将秒数转换成字符串输出
    gettimeofday(&tpend,NULL);
    timeuse=1000000 * (tpend.tv_sec - tpstart.tv_sec) + tpend.tv_usec - tpstart.tv_usec;
//    timeuse /= 1000000;
    printf("      Used Time:%f\n",timeuse);
    }
}
void init_sigaction(void){
```

```
    struct sigaction act;
    act.sa_handler = prompt_info;
    act.sa_flags   = 0;
    sigemptyset(&act.sa_mask);                 //初始化一个空的信号集合 act.sa_mask
    sigaction(SIGALRM, &act, NULL);            //为 SIGALRM 信号设置信号处理函数
    gettimeofday(&tpstart,NULL);
}
void init_time(void){
    struct itimerval val;
    val.it_value.tv_sec = 0;                   //设置间隔定时器的当前值
    val.it_value.tv_usec = 10000;//取值要大等于时钟滴答的周期，否则仍为时钟滴答的时间
    val.it_interval = val.it_value;            //间隔计数器的初始值
    setitimer(ITIMER_REAL, &val, NULL);//设置真实间隔定时器,定时结束后将发出 SIGALRM 信号
}
int main(void){
    printf("clock tick frequency is %d\n",HZ); //输出时钟滴答的频率
    init_sigaction();
    init_time();
//    printf("clock tick frequency is %d\n",HZ);
    while(1);
    exit(0);
}
```

注：单独运行该测试程序时，几乎看不到什么误差，但在运行比较复杂程序的同时，再运行该测试程序，误差就会显示出来，而且比想象中要恶劣的多，所以在实际应用时一定要注意！

二. 虚拟地址

在标准的 Linux 中对硬件物理地址的访问和 uClinux 是完全不同的，在 uClinux 下可以直接引用硬件的物理地址，因为 uClinux 是针对无 MMU 的处理器，因此对硬件物理地址采用的是实际地址。与此相反，标准 linux 下，硬件的实际物理地址是直接访问不到的，必须映射到 Linux 的虚拟地址空间统一管理。如特殊功能寄存器，完成物理地址到虚拟地址转换的函数和定义分别位于 Linux 源码的 kernel/arch/arm/mach-s3c2410/generic.c 和 kernel/include/asm-arm/arch-s3c2410/Hardware.h。

generic.c 中相关程序如下：

```
static struct map_desc standard_io_desc[] __initdata = {
 /* virtual     physical     length      domain     r  w  c  b */
 { 0xe8000000, 0x48000000, 0x17000000, DOMAIN_IO, 0, 1, 0, 0 },
 LAST_DESC
};
void __init s3c2410_map_io(void){
      iotable_init(standard_io_desc);
}
```

Hardware.h 中的相关定义如下：

```
#define VIO_BASE        0xe8000000    /* virtual start of IO space */
#define PIO_START       0x48000000    /* physical start of IO space */
```

如外部设备，完成物理地址到虚拟地址转换的函数和定义分别位于 Linux 源码的 kernel/arch/arm/mach-s3c2410/smdk.c 和 kernel/include/asm-arm/arch-s3c2410/smdk.h。

smdk.c 中相关程序如下：

```
static struct map_desc smdk_io_desc[] __initdata = {
    /* virtual     physical     length      domain     r  w  c  b */
    { vCS8900_BASE, pCS8900_BASE, 0x00100000, DOMAIN_IO, 0, 1, 0, 0 },
    { vCF_MEM_BASE, pCF_MEM_BASE, 0x01000000, DOMAIN_IO, 0, 1, 0, 0 },
    { vCF_IO_BASE, pCF_IO_BASE, 0x01000000, DOMAIN_IO, 0, 1, 0, 0 },
    LAST_DESC
};
static void __init smdk_map_io(void){
     s3c2410_map_io();
     iotable_init(smdk_io_desc);

     s3c2410_register_uart(0, 0);
     s3c2410_register_uart(1, 1);
     set_gpio_ctrl(GPIO_IR_TXD);
     set_gpio_ctrl(GPIO_IR_RXD);
     s3c2410_register_uart(2, 2);

#ifdef CONFIG_PM
      register_wakeup_src(0, EXT_FALLING_EDGE, 0);
#endif
}
```

smdk.h 中的相关定义如下：

```
/* CS8900a, nGCS3 */
#define pCS8900_BASE        0x19000000
#define vCS8900_BASE        0xd0000000

/* PCMCIA, nGCS2 */
#define pCF_MEM_BASE        0x10000000
#define vCF_MEM_BASE        0xd1000000
#define pCF_IO_BASE         0x11000000
#define vCF_IO_BASE         0xd2000000
```

三．以太网控制器 —— CS8900A 硬件调试

1．调试步骤

Step 1： 检查网络的物理连接。

用交叉网线将目标板于 PC 机相联，此时目标板和 PC 机的以太网连接指示灯（LINKLED）将常亮，活动指示灯将闪烁，PC 机（windows 系统）将出现网络已连接信息。

如果不正常，可能是接收或发送链路还有问题，或芯片的该相关部分有问题，特别注意芯片第 93 脚（RES）连接的电阻阻值。

Step 2： 读取芯片 ID 判断芯片及 ISA 总线是否正常。

可在 kernel/drivers/net/cs8900a.c 的初始化部分增加读取芯片 ID 号及打印的代码。正常的 ChipID：630e，通常不正确时读出来 ChipID：dc90。此时可能是 I/O 和内存空间读写部分的逻辑转换电路，或地址数据线不正常等。

也可直接在 ADS 或其它环境下写一小段不用在 linux 下运行的代码：

```
#define CS8900_BASE    (*(volatile unsigned short*)0x19000000)
#define EthIOAddr      (*(volatile unsigned short*)0x19000300)
#define EthPPP         (*(volatile unsigned short*)0x1900030a)
#define EthPPD0        (*(volatile unsigned short*)0x1900030c)
void Test_CS8900(void){
    rBWSCON = rBWSCON & ~(0xf << 12) | (0xd << 12);//nGCS3=nUB/nLB(sSBHE),nWAIT,16-bit
    rBANKCON3 = (0 << 13) | (3 << 11) | (7 << 8) | (1 << 6) | (0 << 4) | (3 << 2) | 0;
    EthPPP = 0;
    Uart_Printf("CS8900A ChipID1 is %x\n", EthPPD0);//打印 ID
    EthPPP = 2;
    Uart_Printf("CS8900A ChipID2 is %x\n", EthPPD0);
}
```

以上两部完成后在启动内核的初始化以太网时将打印下述信息：

Eth0：cs8900 rev K(3.3 Volts) found at 0xd0000300

Cs89x0 media RJ-45, IRQ 37

然后启动系统后就可以用 ifconfig eth0 进行设置，ping 通了。

2．出现过的问题

1> 用交叉网线连接到 PC 机时网络指示灯不亮，PC 机没有任何反应。

原因：CS8900A 第 93 脚连接的 RES 阻值不正确

2> PC 机显示网络已连接（只说明物理层正常），但读取 CS8900A 的 ID 不正确。

原因：总线没有配置好，如：rBWSCON = rBWSCON & ~(0xf << 12) | (0xd << 12); //nGCS3=nUB/nLB(sSBHE),nWAIT,16-bit

rBANKCON3 = (0 << 13) | (3 << 11) | (7 << 8) | (1 << 6) | (0 << 4) | (3 << 2) | 0;

或缓冲芯片(74LVCH162245)等不正常或虚焊

3> 在以太网接口芯片 CS8900A 和 PCMCIA 接口 PD6700 电路中为何都要用到 ADDR24 及一些组合逻辑电路？

原因：CS8900A 和 PD6700 都为 ISA 总线接口，而在 ISA 总线接口中，I/O 空间和内存空间是独立寻址的，因此它们有各自不同的读写信号。但 ARM 体系中，I/O 空间和内存空间是统一寻址，因此电路设计的时候用高位地址线(ADDR24)经过组合逻辑的译码来区分 I/O 空间和内存空间的读写。如使用 nGCS3 的 CS8900A，它的内存空间地址为 0x19000000(ADDR24 = 1)，则 I/O 空间地址为 0x18000000。又如使用 nGCS2 的 PD6700，它的内存空间地址为 0x11000000，则 I/O 空间地址为 0x10000000。

四．WiFi 无线网络

1．在 RedHat9 上安装 TL-WN210 无线网卡驱动

1．1 安装步骤

在调试目标板的无线网络时，PC 机也必须有一个无线网卡和其连接。目前比较便宜通用的是 TP-LINK，但其没有 linux 下的驱动，所以下面介绍用 ndiswrapper 驱动该网卡的方法。

Step 1：**下载ndiswrapper**。从该项目的主页（http://sourceforge.net/projects/ndiswrapper/）下载ndiswrapper-1.1.tar.gz。

Step 2：**重新编译内核**。在安装 ndiswrapper 包之前，要保证/lib/modules/2.4.20-8/build 存在，它是指向/usr/src/linux-2.4.20-8 目录的符号链接。重新编译内核，否则安装 ndiswrapper 过程中，执行“depmod -a”命令时，会出现以下错误：

depmod: *** Unresolved symbols in /lib/modules/2.4.21-166-default/misc/ndiswrapper.o。安装完执行“modeprobe ndiswrapper”命令时也会执行上述的一系列错误，而失败。即使都成功，网卡在运行过程当中也有可能会出不正常现象，如定时的往终端上打印信息的现象。

具体操作如下：切换到/usr/src/linux-2.4.20-8 目录，运行 make mrproper 保证源码树是干净的，查看该目录下的 Makefile，将“EXTRAVERSION = -8custom” 的 custom 字样去掉，运行 make menuconfig，可以不修改，保存配置文件，运行 make dep。

Step 3：**安装 ndiswrapper**。解压缩下载的 ndiswrapper-1.1.tar.gz 文件，切换到压缩后的目录，运行 make clean，再运行 make install 安装。

Step 4：**安装网卡驱动**。将网卡驱动光盘里该网卡在 Windows XP 下的安装信息文件(NET8180.INF)和系统文件(rtl8180.sys)复制到一个目录中(如在当前目录下新建 TL-WN210，将两个文件复制到 TL-WN210 目录)。用“ndiswrapper - I ./TL-WN210/NET8180.INF”命令来加载该网卡 Windows 下的驱动，安装成功后会提示：install NET8180，如果用“ndiswrapper - l”命令会列出加载的驱动程序。最后运行“modprobe ndiswrapper”命令加载 ndiswrapper 模块，此时无线网卡的 POWER 指示灯将闪烁（每次重启后，都需要重新执行“modprobe ndiswrapper”命令加载 ndiswrapper 模块)。如果不闪烁，则执行“/etc/init.d/pcmcia restart”。

Step 5：**使用 iwconfig 设定无线接口**。当网卡的 POWER 指示灯将闪烁后，输入“iwconfig”命令列出可用的无线网络接口，及各接口的操作模式、频率、无线接入点的 MAC 地址、信号质量、电源管理状态、密钥值等。

```
[root@zkccn /]#iwconfig
Lo         no wireless extensions.
eth0       no wireless extensions.
Wlan0  IEEE 802.11b ESSID:off/any
Mode:Auto Frequency:2.412GHz Access Point: 00:00:00:00:00:00
Bit Rate=11Mb/s Tx-Power=20 dBm Sensitivity=0/3
RTS thr:2432 B Fragment thr:2432 B
Encryption key:off
Power Management:off
Link Quality:100/100 Signal level:-80 dBm Noise level:-256 dBm
Rx invalid nwid:0 Rx invalid crypt:0 Rx invalid frag:0
Tx excessive retries:0 Invalid misc:0 Missed beacon:0
```

Step 6：**搜索网络**。可用 iwlist 来扫描邻近无线接入点的信标帧：

```
[root@zkccn /]#iwlist wlan0 scan
```

Wlan0   Scan completed :

Cell 01 - Address: 00:15:E9:DE:B1:5D

ESSID:"R$DNAN6"

(Unknown Wireless Token 0x8B01)

Mode:Master

Frequency:2.422GHz

Quality:0/100 Signal level:-80dBm Noise level:-256 dBm

Encryption key:on

Bit Rate:1Mb/s

Bit Rate:2Mb/s

Bit Rate:5.5Mb/s

Bit Rate:11Mb/s

Extra:bcn_int=100

Extra: atim=0

Step 7：连接无线网络

连接无线网络的第一步，是使用 iwconfig 的 essid 选项让网卡知道你想参与的无线网络的标识（ESSID）：

[root@zkccn /]#iwconfig wlan0 essid "R&DNAN6"

每个无线网络都有一个标识，而且区分大小写。若名称中含有空格，则必须以一对双引号括住。设定网络标识后，还需要该标识的密码（WEP key），否则执行上面指令后还不能连接上（用“iwconfig wlan0”时显示 ESSID：off/any,而不是预期的 ESSID："R$DNAN6"）。设定 key 指令：

[root@zkccn /]#iwconfig wlan0 key B19227EF6E

设定好标识及密码之后，无线网卡的 LINK 指示灯会亮，表明已连接上无线网络的接入点。可以再一次用 iwconfig 检查设定结果：

[root@zkccn /]#iwconfig wlan0

Wlan0 IEEE 802.11b ESSID: "R$DNAN6"

Mode:Managed Frequency:2.412GHz Access Point: 00:15:E9:DE:B1:5D

Bit Rate=11Mb/s Tx-Power=20 dBm Sensitivity=0/3

RTS thr:2432 B Fragment thr:2432 B

Encryption key:B192-27EF-6E Encryption mode:restricted

Power Management:off

Link Quality:100/100 Signal level:-80 dBm Noise level:-256 dBm

Rx invalid nwid:0 Rx invalid crypt:0 Rx invalid frag:0

Tx excessive retries:0 Invalid misc:0 Missed beacon:0

除了 WEP-key 之外，还必须选择 WEP 系统形式。开放式（open）系统接受明文形式的数据帧，而受限（restricted）系统会丢弃明文形式的数据帧。

[root@zkccn /]#iwconfig wlan0 key open

[root@zkccn /]#iwconfig wlan0 key restricted

设定好所要用的 WEP-key 与 WEP 系统形式。

另外，如果为点对点还需设置连接模式，如：

[root@zkccn]#ifconfig wlan0 mode ad-hoc

Step 8：获取 IP 地址，访问网络

当连接上无线接入点后，还不能访问网络，还需配置 IP 地址，用 ifconfig 命令查看状态：

[root@zkccn /]#ifconfig wlan0

Wlan0 Link encap:Ethernet HWaddr 00:0A:EB:A4:DE:A3

BROADCAST MULTICAST MTU:1500 Metric:1

RX packets:14 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:100

RX bytes:14472 (14.1 Kb) TX bytes:4140 (4.0 Kb)

Memory:30800000-30800024

此时如果 PING 会失败，如下：

[root@zkccn /]#ping 192.168.1.99

Connect:Network is unreachable　　　　　　　(网络不能获得)

创建/etc/sysconfig/network-scripts/ifcfg-wlan0 文件，使无线网卡自动获得 IP 地址。该文件内容如下：

BOOTPROTO=‘dhcp’

MTU=‘’

REMOTE_IPADDR=‘’

STARTMODE=‘onboot’

UNIQUE=‘’

编辑完上述文件保存后，用 dhclient 来取得 IP 地址：

[root@zkccn /]#dhclient wlan0

如果不成功，可能是没有 ifcfg-wlan0 文件

[root@zkccn /]#dhclient eth1

Internet Systems Consortium DHCP Client V3.0pl1

Copyright 1995-2001 Internet Systems Consortium.

All rights reserved.

For info, please visit http://www.isc.org/products/DHCP

Listening on LPF/wlan0/00:0a:eb:a4:de:3a

Sending on LPF/wlan0/00:0a:eb:a4:d3:3a

Sending on Socket/fallback

DHCPDISCOVER on wlan0 to 255.255.255.255 port 67

DHCPACK from 192.168.2.1

bound to 192.168.2.163 -- renewal in 601821 seconds.

[root@zkccn /]#ifconfig wlan0

Wlan0 Link encap:Ethernet HWaddr 00:0A:EB:A4:DE:A3

inet addr:192.168.2.163 Bcast:192.168.2.255 Mask:255.255.255.0

UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

RX packets:74 errors:0 dropped:0 overruns:0 frame:0

TX packets:28 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:100

RX bytes:14472 (14.1 Kb) TX bytes:4140 (4.0 Kb)

Memory:30800000-30800024

现在就可以 ping 通公司内部 BBS 了如:

```
[root@zkccn /]# ping 192.168.1.99
PING 192.168.1.99(192.168.1.99)56(84) bytes of data.
64 bytes from 192.168.1.99: icmp_seq=1 ttl=127 time=4.46ms
64 bytes from 192.168.1.99: icmp_seq=2 ttl=127 time=1.68ms
64 bytes from 192.168.1.99: icmp_seq=3 ttl=127 time=2.01ms
--- 192.168.1.99 ping statistics ---
3 packets transmitted, 8 received, 0% packet loss, time 3047ms
Rtt min/avg/max/mdev = 1.681/2.150/4.465/0.882 ms
```

然后进入 X Window 环境，打开浏览器，就可以上网了。

Step 9：删除 ndiswrapper

卸载前首先要用 modprobe -r ndiswrapper 来卸载这个内核模块， 用 ndiswrapper -e drivername 来卸载安装的一个具体驱动程序，删除/etc/ndiswrapper 目录下的关于该驱动程序的文件夹， 就可以运行 make uninstall 命令来卸载程序了。

1．2 出现的问题：

“modprobe ndiswrapper”命令加载 ndiswrapper 模块时，网卡的 POWER 灯不亮，且运行“cardctl eject”命令时提示/proc/driver 目录下找不到 pcmcia 驱动，运行“/etc/rc.d/init.d/pcmcia restart”时出现： …… pcmcia_core.o ……

原因是删除了 pcmcia 模块及相关驱动，需要重装，可以安装 pcmcia-cs-3.2.8.tar。

方法：

到http://pcmcia-cs.sourceforge.net下载pcmcia-cs-3.2.8.tar

```
[root@zkccn src]#tar zxvf pcmcia-cs-3.2.8.tar
[root@zkccn src]#cd pcmcia-cs-3.2.8
[root@zkccn pcmcia-cs-3.2.8]#make config
```

中间有一步需设定 kernel source 所在目录：

```
Linux kernel source directory [/usr/src/linux]: /usr/src/linux-2.4.20-8
[root@zkccn pcmcia-cs-3.2.8]#make all
[root@zkccn pcmcia-cs-3.2.8]#make install
[root@zkccn pcmcia-cs-3.2.8]#/etc/rc.dinit.d/pcmcia restart
```

重新启动 pcmcia 装置

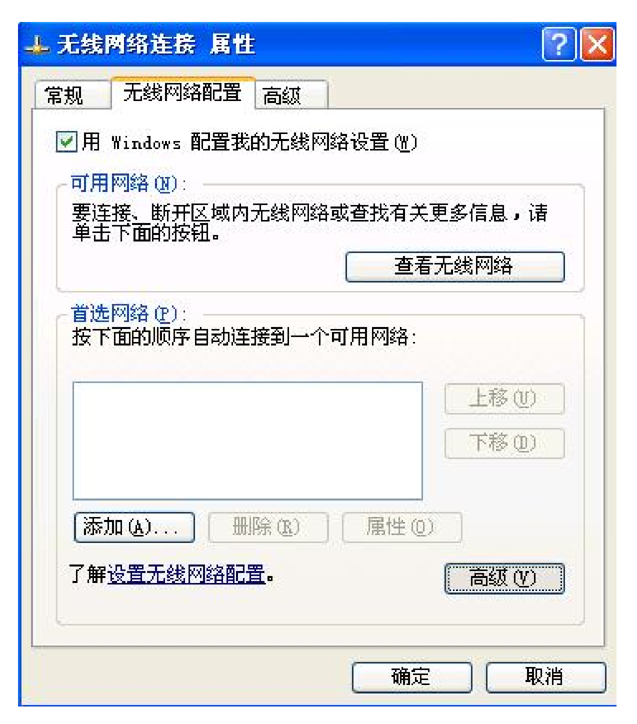
```
[root@zkccn pcmcia-cs-3.2.8]#dump_cis
```

观察是否有抓到网络卡

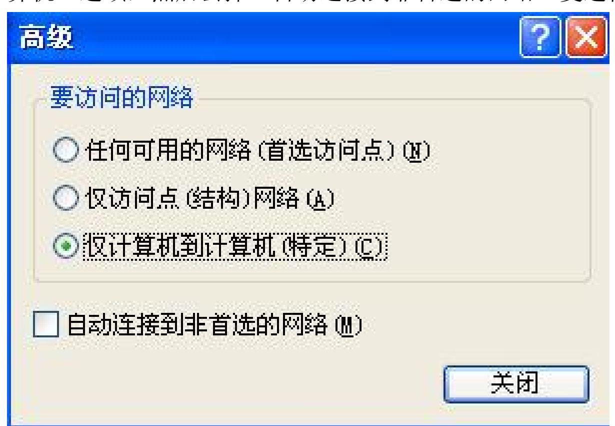
2. 无线网络配置

2．1 Window XP 下配置点对点连接

Setup 1： 打开“无线网络连接”属性对话框，切换到“无线网络配置”选项卡，勾选“用 Windows 配置我的无线网络设置”选项，如下图所示。



Setup 2：点击 Setup 1 图所示的“高级”按钮，将会弹出如下图所示对话框。在其中点选“仅计算机到计算机”选项，然后去掉“自动连接到非首选的网络”复选框，最后单击“关闭”按钮。



Setup 3：返回 Setup 2 所示窗口后再单击“添加”按钮。接着会弹出如下图所示“无线网络属性”设置框。在“网络名（SSID）”中输入无线网络的标识，如“my net”，然后去掉“自动为我提供此密钥”选项，再输入 10 位十六进制的数字密钥（注：如不执行这一步，给网络提供密钥，尽管最后网络能够连接，但也不能 ping 通）。最后点“确定”按钮。

无线网络属性

关联 验证 连接

网络名(SSID)(N): my net

无线网络密钥

此网络要求下列密钥:

网络验证(A): 开放式

数据加密(D): WEP

网络密钥(K): **********

确认网络密钥(O): **********

密钥索引(高级)(X): 1

自动为我提供此密钥(H)

这是一个计算机到计算机(特定的)网络；没有使用无线访问点(C)

确定 取消

提示：SSID 值主要用来区分不同的网络，最多可以设置 32 个字符。只有为两台计算机的无线网卡设置了名称相同的 SSID 值，才能实现互相通信。

Setup 4：切换到“常规”选项卡，双击“Internet 协议（TCP/IP)”，打开“Internet 协议（TCP/IP）属性”对话框。选中“使用下面的 IP 地址”选项，将 IP 地址设置为 192.168.0.1，子网掩码设置为 255.255.255.0。

Internet 协议 (TCP/IP) 属性

常规

如果网络支持此功能，则可以获取自动指派的 IP 设置。否则，您需要从网络系统管理员处获得适当的 IP 设置。

自动获得 IP 地址(O)

使用下面的 IP 地址(S):

IP 地址(I): 192 . 168 . 1 . 1

子网掩码(U): 255 . 255 . 255 . 0

默认网关(D): . . .

自动获得 DNS 服务器地址(B)

使用下面的 DNS 服务器地址(E):

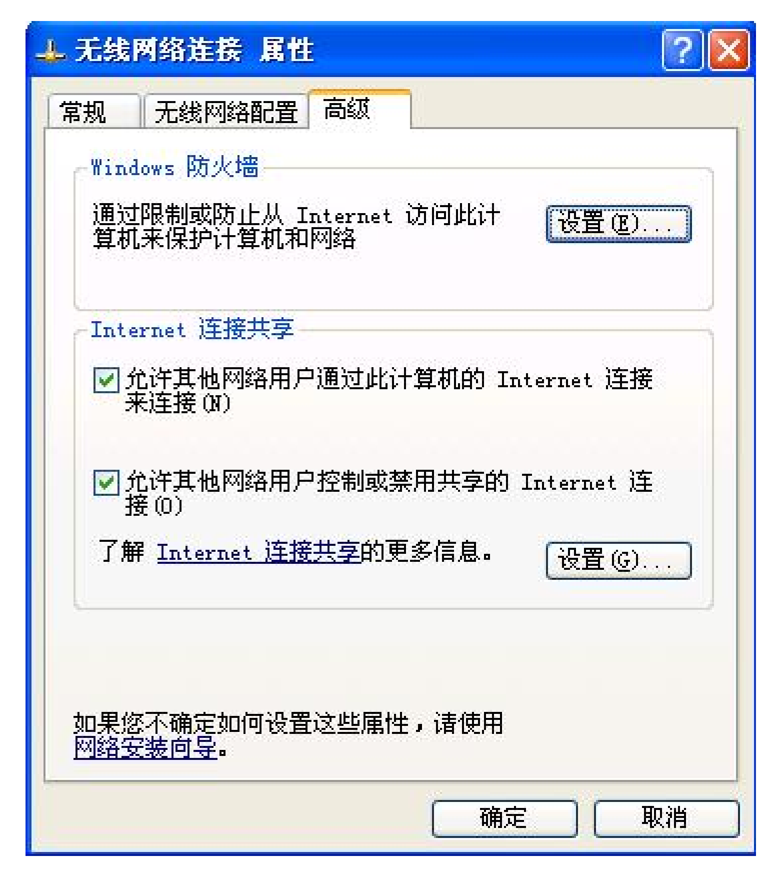
首选 DNS 服务器(P): . . .

备用 DNS 服务器(A): . . .

高级(V)...

确定 取消

Setup 5：切换到“高级”选项卡，然后勾选中 Internet 连接共享的两个选项，如下图所示：



到此完成了主机的设置。

Setup 6：再到另一台计算机进行同样的设置（这里称其为客户机）。主要是 IP 地址一项要注意，应将客户机的 IP 地址设置在与主机同一网段，比如 192.168.0.2；还有网关和 DNS 服务器都应设置为 192.168.0.1（即主机 IP 地址）。

Setup 7：当两台计算机连接到网络后，就可以分别在命令行窗口下 ping 对方的 IP 地址。注：如果网络没有设置密钥时，可能会 ping 不通。

2．2 Linux 下配置点对点连接

Step 1：[root@zkccn /]#modprobe nidswrapper

Step 2：[root@zkccn /]#iwconfig wlan0 mode ad-hoc

Step 3：[root@zkccn /]#iwconfig wlan0 essid “my net”

Step 4：[root@zkccn /]#iwconfig wlan0 key “1234567890”

```
[root@zkccn /]#iwconfig wlan0
Wlan0  IEEE 802.11b ESSID: "my net"
Mode:Ad-hoc Frequency:2.412GHz Access Point: 00:11:22:33:53:B8
Bit Rate=11Mb/s Tx-Power=20 dBm Sensitivity=0/3
RTS thr:2432 B Fragment thr:2432 B
Encryption key:1234-5678-90 Encryption mode:restricted
Power Management:off
Link Quality:100/100 Signal level:-80 dBm Noise level:-256 dBm
Rx invalid nwid:0 Rx invalid crypt:0 Rx invalid frag:0
Tx excessive retries:0 Invalid misc:0 Missed beacon:0
```

Step 5：[root@zkccn]#ifconfig wlan0 192.168.1.1

[root@zkccn]# iwconfig wlan0

Wlan0 Link encap:Ethernet HWaddr 00:0A:EB:A4:DE:A3

inet addr:192.168.1.1 Bcast:192.168.1.255 Mask:255.255.255.0

UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

RX packets:74 errors:0 dropped:0 overruns:0 frame:0

TX packets:28 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:100

RX bytes:14472 (14.1 Kb) TX bytes:4140 (4.0 Kb)

Memory:30800000-30800024

此时，windows 从机将从原来的断开状态变成连接状态（可观察右下角任务栏的无线网络图标）。可以相互 ping 对方的 IP 进行测试网络的连接状态：

Step 6：[root@zkccn]#ping 192.168.1.1

PING 192.168.1.1(192.168.1.1)56(84) bytes of data.

64 bytes from 192.168.1.99: icmp_seq=1 ttl=127 time=4.46ms

64 bytes from 192.168.1.99: icmp_seq=2 ttl=127 time=1.68ms

64 bytes from 192.168.1.99: icmp_seq=3 ttl=127 time=2.01ms

--- 192.168.1.1 ping statistics ---

3 packets transmitted, 8 received, 0% packet loss, time 3047ms

Rtt min/avg/max/mdev = 1.681/2.150/4.465/0.882 ms

3．RedHat9 上使用 WL-110 无线网卡

3．1 安装步骤

Step 1：安装 pcmcia-cs

到http://pcmcia-cs.sourceforge.net下载pcmcia-cs-3.2.8.tar.gz

解压缩：

[root@zkccn wireless_internet]#tar zxvf pcmcia-cs-3.2.8.tar.gz

[root@zkccn wireless_internet]#cd pcmcia-cs-3.2.8

配置：

[root@zkccn pcmcia-cs-3.2.8]#make config

中间有一步需设定 kernel source 所在目录：

Linux kernel source directory [/usr/src/linux]:/usr/src/linux-2.4.20-8

编译：

[root@ zkccn pcmcia-cs-3.2.8]#make all

安装：

[root@ zkccn pcmcia-cs-3.2.8]#make install

重新启动 pcmcia 装置

[root@ zkccn pcmcia-cs-3.2.8]#/etc/rc.d/init.d/pcmcia restart

观察是否有抓到网卡

[root@ zkccn pcmcia-cs-3.2.8]#dump_cis

Step 2：安装 linux-wlan-ng

安装 linux-wlan-ng 之前需重新配置内核，去掉不正确的目标文件及关联（具体操作：切换到/usr/src/linux-2.4.20-8 目录，运行 make mrproper 保证源码树是干净的，查看该目录下的 Makefile，将“EXTRAVERSION = -8custom” 的 custom 字样去掉，运行 make menuconfig，可以不修改，保存配置文件，运行 make dep)，否则在 linux-wlan-ng 目录下，make all 时会出现错误而中止编译。

到http://www.linux-wlan.com/linux-wlan/ 下载linux-wlan-ng-0.2.0.tar.gz

解压缩：

[root@zkccn wireless_internet]#tar zxvf linux-wlan-ng-0.2.0.tar.gz

配置：

[root@zkccn wireless_internet]#cd linux-wlan-ng-0.2.0

[root@zkccn linux-wlan-ng-0.2.0]#make clean

注：一定要要执行该步（make clean)，否则安装成功后，重启 PCMCIA 服务后还是不能成功，lsmod 显示模块时，也没有看到 prism2_cs、p80211、ds 等模组的载入。

[root@zkccn wireless_internet]#make config

中间有一步需设定 kernel source 所在目录：

  Linux kernel source directory [/usr/src/linux]:/usr/src/linux-2.4.20-8

编译：

[root@zkccn linux-wlan-ng-0.2.0]#make all

安装：

[root@ zkccn wireless_internet]#make install

Step 3：更改/etc/modules.conf

[root@zkccn root]#vi /etc/modules.conf

档案中新增一行以表明当开机时（或 wlan0 启用时）要去载入哪一个驱动程式模组

alias wlan0 prism2_cs

Step 4：配置无线网络：

/etc/wlan/wlan.conf ：指定 ESSID。

This file maps between wlan devices and network IDs, and contains the names of all devices that should be initialized by the hotplug and rc scripts.

/etc/wlan/wlancfg-* ：设定 WEP、mode 等

These files are per-network configurations. This makes it easy to switch between different SSIDs and the various settings they may require, like WEP keys and whatnot.

0> This example assumes your network name/SSID is "MyHomeNetwork"

1> cp /etc/wlan/wlancfg-DEFAULT /etc/wlan/wlancfg-MyHomeNetwork

2> edit /etc/wlan/wlan.conf and change the SSID_wlan0 line to:

 SSID_wlan0="MyHomeNetwork"

3> edit /etc/wlan/wlancfg-MyHomeNetwork, and make any necessary changes
    necessary to support your network, such as WEP and whatnot.

/etc/pcmcia/network.opts：设置 IP 地址等

Step 5：使用 wlanctl-ng 工具配置无线网络：

Linux-wlan-ng 驱动不能用 wireless-tools 系列的工具（如 iwconfig）来配置，而必须用专门的配置工具 wlanctl-ng。

[root@zkccn root]#wlanctl-ng wlan0 lnxreq_autojoin ssid=“my net” authtype=opensystem

Step 6：用脚本配置无线网络

1> **配置网卡**。主要是/etc/wlan 目录下创建的 wlancfg-essid 配置 WEP，MODE 等。wlan.conf 文件使能及使用哪个 SSID，如：SSID_wlan0=”R&DNAN3”是，则 SSID 为 R&DNAN3，上述也只能是 wlancfg-R&DNAN3 文件有效。该文件主要设置如下：

Lnxreq_hostWEPEncrypt = true

Lnxreq_hostWEPDecrypt = true

Dot11PrivacyInvoked = true

Dot11WEPDefaultKeyID = 0

Dot11ExcludeUnencrypted = true

Dot11WEPDefaultKey0=B1:92:27:EF:6E

IS_ADHOC=n

AuthType=”opensystem”

其它默认

上述为 AP 模式，如果想配置成点对点通信，则上述设置成“IS_ADHOC=y”即可。

2> **配置网络**。/etc/pcmcia/network.opts 主机设置 IP 地址，网关等。如果是自动分配地址，则只需将 DHCP = “y”，其它的如 IP 地址、广播地址、网关等设成空就可以。

启动过程中设置 IP 的方法：创建/etc/sysconfig/network-scripts/ifcfg-wlan0 文件。

Get IP setting from DHCP：

DEVICE=’wlan0’

BOOTPROTO=’dhcp’

ONBOOT=’yes’

Fixed Setting：

DEVICE=’wlan0’

IPADDR=’192.168.2.98’

NETMASK=’255.255.255.0’

NETWORK=’192.168.2.0’

BROADCAST=’192.168.2.255’

ONBOOT=’yes’

GATEWAY=’192.168.2.254’

Step 8：重启 PCMCIA : #/etc/init.d/pcmcia restart

查看结果：#iwconfig wlan0

配置 IP：#ifconfig wlan0 192.168.2.1

注意：如果设置的 IP 地址已被附近其它无线网络占用了，此时应修改，否则即时连接成功，也无法 ping 通。

3．2 出现的问题：

启动 PCMCIA 出现错误

[root@ zkccn root]#/etc/rc.d/init.d/pcmcia restart

Shutting down PCMCIA services：cardmgr modules.

Starting PCMCIA services：modulesmodprobe：Can’t locate module pcmcia_core.o

Modprobe：Can’t locate module yenta_socket.o

Modprobe：Can’t locate module ds.o

[root@netlab38 root]#cardctl eject

No pcmcia driver in /proc/devices

上述错误是没有执行 Step 1 安装 pcmcia-cs-3.2.8.tar.gz 引起， 且在重新编译内核时不能去掉PCMCIA 相关选项（《Linux 的无线连接教程》中提到要去掉） 。

3．3 wlanctl-ng 命令

3．3．1 命令说明

1> Loading module

]$modprobe prism2_cs

2> reload the firmware

]$wlanctl-ng wlan0 lnxreq_ifstate ifstate=disable

]$wlanctl-ng wlan0 lnxreq_ifstate ifstate=enable

3> Set PortType

]$wlanctl-ng wlan0 dot11req_mibset mibattribute=p2CnfPortType=Type

Type：   0 for IBSS mode (Ad-hoc)

1 for BSS mode (Infrastructure)

*Note that the set port type should be done before auto join command

4> Auto join command

]$wlanctl-ng wlan0 lnxreq_autojoin ssid=<SSID> authtype=Type

Type：   opensystem

sharedkey

5> Set Channel

]$wlanctl-ng wlan0 dot11req_mibset mibattribute= p2CnfOwnChannel=Value

The Value should be in the range 1~14。

6> Set RTSThreshold

]$wlanctl-ng wlan0 dot11req_mibset mibattribute=dot11RTSThreshold=Value

The Value should be in the range 0~2347

7> Set FragmentThreshold

]$wlanctl-ng wlan0 dot11req_mibset mibattribute=dot11FragmentThreshold =Value

The Value should be in the range 256~2346

8> Set SSID

]$wlanctl-ng wlan0 dot11req_mibset mibattribute= p2CnfOwnSSID=<your ssid>

9> Set WEP Key

]$wlanctl-ng wlan0 dot11req_mibset mibattribute=dot11WEPDefaultKeyID=<KeyIndex>

]$wlanctl-ng wlan0 dot11req_mibset mibattribute=dot11ExcludeUnencrypted=true

]$wlanctl-ng wlan0 dot11req_mibset mibattribute=dot11PrivacyInvoked=true

]$wlanctl-ng wlan0 dot11req_mibset mibattribute=dot11WEPDefaultKey<KeyIndex>=xx:xx:xx..

*KeyIndex should be in the range of 0~3

** xx:xx:xx… is the WEP key. xx is in the hexdecimal format. If you want to use WEP 64, 5 hexdecimal values are expected. 13 hexdecimal values are expected when using WEP 128.

The four commands listed above are the basic commands you have to issue when enabling WEP and setting WEP key. One can set four WEP keys. If you want to set four keys at the same time, you

can repeat the last command with changing the KeyIndex.

After setting the WEP key information, one can join to an Access Point by the following autojoin command.

]$ wlanctl-ng wlan0 lnxreq_autojoin ssid=<SSID> authtype=sharedkey

10> Site Survey

]$ wlanctl-ng wlan0 dot11req_scan bsstype=any bssid=00:00:00:00:00:00 \
scantype=both probedelay=0 channellist="01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:00" \
minchanneltime=200 maxchanneltime=1000 ssid=""

]$ wlanctl-ng wlan0 dot11req_mibget mibattribute=p2CommunicationScanresult

It includes two commands when doing site survey. The first command is issue a scanresult command to the firmware which asks firmware to scan the channel. The second one is display the scan
result returned by firmware.

Site survey is very useful when searching the APs. One may want to join the AP with the specific SSID or join the AP which has the best signal.

11> Get Tally information

]$ wlanctl-ng wlan0 dot11req_mibget mibattribute=p2CommunicationTallies

3.3.2 命令举例

例子 1：

```
modprobe prism2_usb prism2_doreset=1
wlanctl-ng wlan0 lnxreq_ifstate ifstate=enable
wlanctl-ng wlan0 lnxreq_autojoin ssid=WLAN authtype=opensystem
wlanctl-ng wlan0 lnxreq_hostwep decrypt=true encrypt=true
wlanctl-ng wlan0 dot11req_mibset mibattribute=dot11WEPDefaultKeyID=0
wlanctl-ng wlan0 dot11req_mibset mibattribute=dot11WEPDefaultKey0=xx:xx:xx
ifconfig wlan0 192.168.42.42
route add default gw 192.168.42.1
ping -c 4 192.168.42.1
```

例子 2：

```
#!/bin/bash
wlanctl-ng wlan0 lnxreq_ifstate ifstate=enable
wlanctl-ng wlan0 dot11req_mibset mibattribute=p2CnfRoamingMode=1
wlanctl-ng wlan0 dot11req_mibset mibattribute=dot11WEPDefaultKeyID=0
wlanctl-ng wlan0 dot11req_mibset mibattribute=dot11ExcludeUnencrypted=true
wlanctl-ng wlan0 dot11req_mibset mibattribute=dot11PrivacyInvoked=true
wlanctl-ng wlan0 dot11req_mibset
mibattribute=dot11WEPDefaultKey0=00:00:00:00:00:00:00:00:00:00:00:00:00
wlanctl-ng wlan0 lnxreq_autojoin ssid="somessid" authtype=sharedkey
ifconfig eth0 down
ifconfig wlan0 192.168.0.163 up
route add default gw 192.168.0.1
```

例子 3：

```
#! /bin/sh
#
```

```
# Wireless USB setup
#
# Step 1 - enable wireless USB for wlan0
wlanctl-ng wlan0 lnxreq_ifstate ifstate=enable
# Step 2 - set SSID for your network
wlanctl-ng wlan0 lnxreq_autojoin ssid=stayoutofmynet authtype=opensystem
# Step 3 - set WEP attributes
wlanctl-ng wlan0 lnxreq_hostwep encrypt=true decrypt=true
wlanctl-ng wlan0 dot11req_mibset mibattribute=dot11PrivacyInvoked=true
# Step 4 - set WEP key
wlanctl-ng wlan0 dot11req_mibset mibattribute=dot11WEPDefaultKeyID=3
wlanctl-ng wlan0 dot11req_mibset mibattribute=dot11WEPDefaultKey3=12:34:56:78:9A
# Step 5 - set IP configuration
ifconfig wlan0 192.168.1.100 netmask 255.255.255.0 broadcast 192.168.1.255
route add default gw 192.168.1.11
```

4. 无线网卡控制器 PD6710 硬件测试

由于 PD6710 与 CS8900A 都为 ISA 接口，所以当以太网控制器 CS8900A 调试成功后，再调试 PD6710 基本上很容易了。测试程序可以参考《32 位嵌入式系统硬件设计与调试》一书中的 PD6710 部分，可在 ADS1.2 上编译硬件仿真或下载测试。它首先检测 PD6710 工作及对它的访问是否正常，如果正常，将在终端中打印"PD6710 hardware is found!!"。其次检测 PCMCIA 网卡是否插入，卡的电压等，打印"Card is found. 3.3V card is detected."。最后对卡进行初始化，读取卡 CIS 信息及打印等。

程序清单：

```
#include "2410ADDR.h"
#include  "def.h"
#define CF_IO_BASE                 0x11000000
#define rPD6710_INDEX              (*(volatile U8 *)(CF_IO_BASE + 0x3E0))
#define rPD6710_DATA               (*(volatile U8 *)(CF_IO_BASE + 0x3E1))
#define CHIP_INFO                  0x1f      /* Chip information */
#define POWER_CTRL                 0x02      /* Power and RESETDRV control */
#define INT_GENERAL_CTRL           0x03      /* Interrupt and general control */
#define MANAGEMENT_INT_CONFIG      0x05      /* Card status change interrupt control */
#define SYS_MEM_MAP0_START_L       0x10      /* System Memory Map 0 Start Address Low */
#define SYS_MEM_MAP0_START_H       0x11      /* System Memory Map 0 Start Address High */
#define SYS_MEM_MAP0_END_L         0x12      /* System Memory Map 0 End Address Low */
#define SYS_MEM_MAP0_END_H         0x13      /* System Memory Map 0 End Address High */
#define CARD_MEM_MAP0_OFFSET_L     0x14      /* Card Memory Map 0 Offset Address Low */
#define CARD_MEM_MAP0_OFFSET_H     0x15      /* Card Memory Map 0 Offset Address High */
#define MAPPING_ENABLE             0x06      /* Address window enable */
#define MISC_CTRL1                 0x16      /* Misc control 1 */
#define MISC_CTRL2                 0x1E      /* Misc control 2 */
#define FIFO_CTRL                  0x17      /* FIFO control */
```

```
#define SETUP_TIMING0            0x3A     /* Setup Timing 0 */
#define CMD_TIMING0              0x3B     /* Command Timing0  */
#define RECOVERY_TIMING0         0x3C     /* Recovery Timing0 */
#define INTERFACE_STATUS         0x01     /* Interface status */
/*******************************************************************************
函数原形：void PD6710_Wr(U8 index, U8 data)
功 能：写 PD6710 寄存器
参 数：
*******************************************************************************/
void PD6710_Wr(U8 index, U8 data){
    rPD6710_INDEX = index;
    rPD6710_DATA = data;
}

/*******************************************************************************
函数原形：U8 PD6710_Rd(U8 index)
功 能：读 PD6710 寄存器
参 数：
*******************************************************************************/
U8 PD6710_Rd(U8 index){
    rPD6710_INDEX = index;
    return rPD6710_DATA;
}

/*******************************************************************************
函数原形：int PD6710_Init(void)
功 能：初始化 PD6710 控制器
参 数：
*******************************************************************************/
int PD6710_Init(void){
    static U8 ChipInformation1,ChipInformation2;
    rGPFCON = rGPFCON & ~(3 << 6) | (2 << 6);
    rGPGCON = rGPGCON & ~(3 << 0) | (2 << 0);
    rBWSCON = rBWSCON & ~(0xf << 8) | (0xd << 8);     //nGCS2=nUB/nLB(sSBHE),nWAIT,16-bit
    rBANKCON2 = (0 << 13) | (3 << 11) | (7 << 8) | (1 << 6) | (0 << 4) | (3 << 2) | 0;
    PD6710_Wr(CHIP_INFO, 0x0);
    /*PD6710 芯片有一个 CHIP_INFO 寄存器(INDEX 为 0x1F),这个寄存器的最高 2 位被写入后的第一
 次读取会返回 11,第二次读取会返回 00,用这种方法检测总线上是否存在 PD6710 芯片*/
    if ((((ChipInformation1 = PD6710_Rd(CHIP_INFO)) & 0xc0) != 0xc0) || (((ChipInformation2
 = PD6710_Rd(CHIP_INFO)) & 0xc0) != 0x00)){
        Uart_Printf("PD6710 hardware identification error!!!\n");
        Uart_Printf("PD6710  ChipInformation1  is  %x,  ChipInformation2  is  %x\n",
 ChipInformation1, ChipInformation2);
        return 0;
```

```
    }
    Uart_Printf("PD6710 ChipInformation1 is %x, ChipInformation2 is %x\n", ChipInformation1, ChipInformation2);
    Uart_Printf("PD6710 hardware is found!!!\n");
    PD6710_Wr(POWER_CTRL, (0 << 7) | (1 << 5) | (0 << 4) | (0 << 0));//设置 PowerControl 寄存器,Auto Power
    PD6710_Wr(INT_GENERAL_CTRL, (1 << 7) | (0 << 5) | (1 << 4) | (3 << 0));//设置中断寄存器,IRQ3,Enable Manage IRQ,Ring Indicate Enable
    PD6710_Wr(MANAGEMENT_INT_CONFIG, (3 << 4) | (1 << 3) | (1 << 0));//Battery Dead or Status Change,Card Detect Change,IRQ3
    //设置属性内存空间映射系统地址为 0,大小 64KB,偏移量地址为 0,数据宽度为 8 位
    PD6710_Wr(SYS_MEM_MAP0_START_L, 0x0);              //MEM0=8bit data width
    PD6710_Wr(SYS_MEM_MAP0_START_H, 0x0);
    PD6710_Wr(SYS_MEM_MAP0_END_L, 0xf);                //0x0 ~ 0xffff
    PD6710_Wr(SYS_MEM_MAP0_END_H, 0x0 | (1 << 6));     //选用 Timing Set 0
    PD6710_Wr(CARD_MEM_MAP0_OFFSET_L, 0x0);
    PD6710_Wr(CARD_MEM_MAP0_OFFSET_H, 0x0 | (1 << 6)); //nREG 有效,属性内存
    PD6710_Wr(MAPPING_ENABLE, 0x1 | (1 << 6));         //使能内存映射
    //设置 Misc Control 寄存器
    PD6710_Wr(MISC_CTRL1, (0 << 7) | (1 << 4) | (1 << 3) | (1 << 2) | (1 << 1));//3.3V
    PD6710_Wr(MISC_CTRL2, 0x1 | (1 << 4));             //Driver LED
    PD6710_Wr(FIFO_CTRL, 0x80);                        //清空 FIFO
    //缺省访问周期为 300ns 设置 Timing Set 0
    PD6710_Wr(SETUP_TIMING0, 0x2);                     //80ns
     PD6710_Wr(CMD_TIMING0, 0x8);                      //320ns
    PD6710_Wr(RECOVERY_TIMING0, 0x2);                  //80ns
    return 1;
}
/*******************************************************************************
函数原形：void PD6710_Modify(U8 index, U8 mask, U8 data)
功   能：用来修改寄存器中某一个特定位
参   数：
*******************************************************************************/
void PD6710_Modify(U8 index, U8 mask, U8 data){
    rPD6710_INDEX = index;
    rPD6710_DATA = (rPD6710_DATA) &~ mask | data;
}
/*******************************************************************************
函数原形：void PD6710_CardInit(void)
功   能：卡初始化
参   数：
*******************************************************************************/
void PD6710_CardInit(void){
    if (PD6710_Rd(MISC_CTRL1) & 0x1){                    //判断卡是 5V 还是 3.3V 的
```

```
        PD6710_Modify(MISC_CTRL1, 0x2, 0x0);               //5V 供电
        Uart_Printf("5.0V card is detected.\n");
    }
    else{
        PD6710_Modify(MISC_CTRL1, 0x2, 0x2);               //3.3V 供电
        Uart_Printf("3.3V card is detected.\n");
    }
    PD6710_Modify(POWER_CTRL, (1 << 4) | 3, (1 <<4 ) | 1);//设定 Vpp=Vcc
    Delay(100);                                         //延时 10ms 时电源稳定
    PD6710_Modify(INT_GENERAL_CTRL, (1 << 6), 0);         //准备 RESET 信号
    PD6710_Modify(POWER_CTRL, (1 << 7), (1 << 7));        //使能 PCMCIA 卡
    PD6710_Modify(INT_GENERAL_CTRL, (1 << 6), 0);         //Reset PCMCIA 卡
    Delay(1);                                           //Reset 时间应该至少 10us
    PD6710_Modify(INT_GENERAL_CTRL, (1 << 6), (1 << 6));  //结束 Reset
    Delay(200);                                         //等待 20ms
    while (!(PD6710_Rd(INTERFACE_STATUS) & 0x20));        //等待 READY 信号
}
#define Card_RdAttrMem(off)        *((volatile U8 *)(0x10000000 + off))
#define Card_WrAttrMem(off, val)   *((volatile U8 *)(0x10000000 + off)) = val
/*************************************************************************************
函数原形：void PrintCIS(void)
功    能：读取无线网卡信息(CIS)
参    数：
*************************************************************************************/
void PrintCIS(void){
    int i,j;
    int cisEnd = 0;
    static U8 str[16],c;
    Uart_Printf("[Card Information Structure]\n\r");
    while (1){                                            //搜寻 CIS 的结束地址
        if (Card_RdAttrMem(cisEnd * 2) == 0xff)  break;     //0xff = termination tuple
        cisEnd++;
        cisEnd += Card_RdAttrMem(cisEnd *2) + 1;
    }
    Uart_Printf("cisEnd = 0 ~ %x\n\r", cisEnd);             //打印 CIS 信息
    for (i = 0; i < 2 * cisEnd; i += 2){
        c = Card_RdAttrMem(i);
        str[(i % 0x20) / 2] = c;
        Uart_Printf("%2x,",c);
        if ((i % 0x20) >= 0x1e){
            Uart_Printf("//");
            for (j = 0; j < 0x10; j++){
                if ((str[j] >= 0) && (str[j] <= 127)){
                    Uart_Printf("%c", str[j]);
```

```
                }
                else{
                    Uart_Printf(".");
                }
            }
            Uart_Printf("\n\r");
        }
    }
    Uart_Printf("\n\r\n\r\n\r");
}
```

```
/************************************************************************************
函数原形：void Test_PD6710(void)
功能：PCMCIA 无线网卡控制器 PD6710 测试程序
参数：
************************************************************************************/
void Test_PD6710(void){
    Uart_Printf("[PD6710 test for reading pc_card CIS]\n");
    if (!PD6710_Init()) return;
    if ((PD6710_Rd(INTERFACE_STATUS) & 0xc) == 0xc) {      //检查 CD#信号,看是否有卡插入
        Uart_Printf("Card is found.\n");
        Delay(2000);
        PD6710_CardInit();
        PrintCIS();
    }
    else{
        Uart_Printf("Card is not found.\n");
    }
}
```

运行结果：

```
[PD6710 test for reading pc_card CIS]
PD6710 ChipInformation1 is d8, ChipInformation2 is 18
PD6710 hardware is found!!!
Card is not found.
```

插入 ASUS WL-110 CF 无线网卡后的结果：

```
[PD6710 test for reading pc_card CIS]
PD6710 ChipInformation1 is d8, ChipInformation2 is 18
PD6710 hardware is found!!!
Card is found.
3.3V card is detected.
[Card Information Structure]
cisEnd = 0 ~ 95
 0,  3,  0,  0,ff,17,  4,67,5a,  8,ff,1d,  5,  3,67,5a,     //└.┤┘g. │└gZ
 8,ff,15,2a,  5,  0,41,53,55,53,  0,38,30,32,5f,31,  //.┴*|ASUS802_1
31,42,5f,43,46,5f,43,41,52,44,5f,32,35,  0,56,65,//1B_CF_CARD_25Ve
```

```
72,73,69,6f,6e,20,30,31,2e,30,30,  0,  0,ff,20,  4,  //rsion 01.00. ┘
aa,  2,  2,  0,21,  2,  6,  0,22,  2,  1,  7,22,  5,  2,40,    //.┐┐!┐-″┐┌●″|┐@
42,  f,  0,22,  5,  2,80,84,1e,  0,22,  5,  2,60,ec,53,   //B☼″|┐..-″|┐`.S
 0,22,  5,  2,c0,d8,a7,  0,22,  2,  3,  7,22,  8,  4,  6,    //″|┐...″┐└●┘-
ff,ff,ff,ff,ff,ff,22,  2,  5,  1,1a,  7,  3,  1,e0,  3,            //......″┐|┌→●└┌.└
 0,  0,  1,1b,10,c1,  1,19,77,b5,1e,35,b5,3c,36,36,//┌←┼.┌├w.-5.<66
 5,46,7f,ff,ff,
```

5．Linux 下驱动程序及装载

5．1 PCMCIA 驱动

PD6710 兼容经典的 i82365 芯片，不仅 Linux 直接支持 i82365，而且 Linuette 在内核源码中提供了在 i82365 基础上针对 SMDK2410 修改好的源码（位于 kernel/drivers/pcmcia/目录下的 i82365.c 和 i82365.h），所以我们可以直接使用。如果使用带 GUI 的文件系统，则在 usr/lib/modules/2.4.18-rmk7-pxa1/kernel/drivers/pcmcia 目录下提供了编译好的 ds.o、i82365.o、pcmcia_core.o 三个 PCMCIA 接口芯片 PD6710 所需的驱动目标文件。usr/etc/rc.local 启动脚本中的“/etc/init.d/pcmcia start”命令用于启动装载它们，所以在系统启动带 GUI 的文件系统时，会打印出装载信息。如果使用不带 GUI 的 root 文件系统，需要自己拷贝该三个文件，也可以重新编译内核生成。

#cd /linuette/target/box/kernel

#make menuconfig

Load an Alternate Configuration File “arch/arm/def-configs/smdk2410”，保持默认配置即可，此时 pcmcia 接口驱动会以模块形式编译。

#make modules

#ls drivers/pcmcia

此时将能看到 ds.o、i82365.o、pcmcia_core.o 已生成

#make modules_install

将在 /tmp/lib 目录下生成 modules 目录，上述三个文件位于 /tmp/lib/modules/2.4.18-rmk7-pxa1/kernel/drivers/pcmcia 目录下。

将模块打包，准备更新文件系统模块

#cd /tmp/lib/modules/2.4.18-rmk7-pxal

#tar zcvf mod.tgz kernel pcmcia

5．2 装载步骤

目标板平台为 mizi 的 linux 和只读根文件系统 root.cramfs。

Step 1： 在根文件系统 root 目录下创建 usr/lib/modules/2.4.18-rmk7-pxa1， 再在 2.4.18-rmk7-pxa1 目录下创建 modules.dep 文件和 pcmcia 目录（或直接解压上述的 mod.tgz：#tar zxvf /tmp/lib/modules/2.4.18-rmk7-pxa1/mod.tgz -C .），pcmcia 目录存 ds.o、pcmcia_core.o、i82365.o。modules.dep 文件内容如下：

/lib/modules/wireless/p80211.o:

/lib/modules/wireless/prism2_cs.o:

/lib/modules/wireless/prism2_usb.o:

Step 2： 在 usr/lib/modules 下创建 wireless/目录，然后将 linux-wlan-ng-0.20 编译生成的 p80211.o、prism2_usb.o、prism2_cs.o 文件复制到 usr/lib/modules/wireless/文件夹下，并将 wlanctl 文件和 wlan_ad 脚本复制到/sbin/文件夹下。

Step 3：在/etc/pcmcia 目录下放入 config 脚本文件，它是一张表。当 cardmgr 检测到插入的卡时，根据卡的名称来从 config 中查寻所对应的驱动程序。Config 内容清单：

```
device "prism2_cs"  module "prism2_cs"
card "ASUS WL-110 802.11b WLAN CF Card"
    version "ASUS", "802_11B_CF_CARD_25"
    bind "prism2_cs"
# Include configuration files for add-on drivers
source ./*.conf
# Include local configuration settings
```

Step 4：用 mkcramfs 重新生成 root.cramfs 下载到目标板，如果提示文件太大不能下载时，可删除一些无用的文件，或修改 vivi 源码。

Step 5：启动系统后，分别用 insmod 加载 pcmcia_core.o、i82365.o、ds.o 和 p80211.o 模块，并运行 cardmgr 程序。

Step 6：将 CF 无线网卡插入 PCMCIA 接口，将打印出正确信息

Step 7：运行 wland_ad 脚本，内容如下：

```
#!/bin/sh
wlanctl wlan0 lnxreq_ifstate ifstate=disable
wlanctl wlan0 lnxreq_ifstate ifstate=enable
wlanctl wlan0 dot11req_start ssid=linux-wlan bsstype=independent beaconperiod=100 dtimperiod=3 cfpollable=false cfpollreq=false cfpperiod=3 cfpmaxduration=100 probedelay=100 dschannel=6 basicrate1=2 basicrate2=4 operationalrate1=2 operationalrate2=4 operationalrate3=11 operationalrate4=22
ifconfig eth0  down
ifconfig wlan0 192.168.0.2 netmask 255.255.0.0 broadcast 192.168.0.255
```

无误后，用 ifconfig 命令将列出下述信息，且在 PC 机端将连接上 linux-wlan 网络，但仍不能 ping 通。

```
# ifconfig wlan0
wlan0     Link encap:Ethernet  HWaddr 00:11:2F:73:B7:FF
          inet addr:192.168.0.2  Bcast:192.168.0.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:54 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:7448 (7.2 kb)  TX bytes:0 (0.0 b)
          Interrupt:36
```

必须给网络增加网络密钥(WEP Key)后，才能 ping 通，所以上述脚本在 ifconfig eth0 down 上面增加如下内容：

```
wlanctl wlan0 dot11req_mibset mibattribute=dot11WEPDefaultKeyID=0
wlanctl wlan0 dot11req_mibset mibattribute=dot11ExcludeUnencrypted=true
wlanctl wlan0 dot11req_mibset mibattribute=dot11PrivacyInvoked=true
wlanctl wlan0 dot11req_mibset mibattribute=dot11WEPDefaultKey0=12:34:56:78:90
```

在PC机端连接该网络时也用该WEP KEY：1234567890

5．3 装载过程中终端显示信息

整个装载过程当中的终端显示信息如下：

```
#cd /usr/lib/modules/2.4.18-rmk7-pxa1;ls
ds.o           pcmcia_core.o  i82365.o
#insmod pcmcia_core.o
Linux Kernel Card Services 3.1.22
  options:  none
cs.c 1.279 2001/10/13 00:08:28 (David Hinds)
#insmod i82365.o
Intel PCIC probe:
  Cirrus PD6710 ISA-to-PCMCIA at port 0x3e0 ofs 0x00, 1 socket
    host opts [0]: [ring] [dyn mode]
cs: register_ss_entry(1, 0xc4852214)
cs: pcmcia_register_socket(0xc4852214)
#insmod ds.o
ds.c 1.112 2001/10/13 00:08:28 (David Hinds)
cs: bind_device(): client 0xc03fe660, sock 0, dev Driver Services
cs: setup_socket(c3ed4800): applying power
cs: resetting socket c3ed4800
cs: reset done on socket c3ed4800
cs: send_event(sock 0, event 4, pri 0) ZLGMCU
cs2: client : c03fe660
cs2: client->next : 0
cs: register_client(): client 0xc03fe660, sock 0, dev Driver Services
ds: ds_event(0x000004, 0, 0xc03fe660)
#cd ../../wireless;ls
p80211.o      prism2_cs.o   prism2_usb.o
#insmod p80211.o
#lsmod
Module                  Size  Used by
p80211                 18320   0  (unused)
ds                      8880   0  (unused)
i82365.o                9776   1
pcmcia_core            42912   0  [ds i82365.o]
#cardmgr
cardmgr[38]: starting, version is 3.1.22
ds_open(socket 0)
ds_open(socket 1)
ds_open(socket 1)
cardmgr[38]: watdhing 1 sockets
s_ioctl(socket 0, 0x80146401, 0xbffffbf8)
```

```
ds_poll(socket 0)
ds_read(socket 0)
cardmgr[38]: inidializing socket 0
s_ioctl(socket 0, 0x8004640b, 0xbffff934)
cs: read_cis_mem(1, 0x0, 2)
cs:  0x01 0x03 0x3c 0x07 ...
cs: read_cis_mem(1, 0x5, 2)
cs:  0x17 0x04 0x00 0x00 ...
cs: read_cis_mem(1, 0xb, 2)
cs:  0x1d 0x05 0x00 0x00 ...
cs: read_cis_mem(1, 0x12, 2)
cs:  0x15 0x2a 0x00 0x00 ...
cs: read_cis_mem(1, 0x3e, 2)
cs:  0x20 0x04 0x00 0x00 ...
cs: read_cis_mem(1, 0x40, 4)
cs:  0xaa 0x02 0x02 0x00 ...
cs: read_cis_mem(1, 0x44, 2)
cs:  0x21 0x02 0x00 0x00 ...
cs: read_cis_mem(1, 0x48, 2)
cs:  0x22 0x02 0x00 0x00 ...
cs: read_cis_mem(1, 0x4c, 2)
cs:  0x22 0x05 0x00 0x00 ...
cs: read_cis_mem(1, 0x53, 2)
cs:  0x22 0x05 0x00 0x00 ...
cs: read_cis_mem(1, 0x5a, 2)
cs:  0x22 0x05 0x00 0x00 ...
cs: read_cis_mem(1, 0x61, 2)
cs:  0x22 0x05 0x00 0x00 ...
cs: read_cis_mem(1, 0x68, 2)
cs:  0x22 0x02 0x00 0x00 ...
cs: read_cis_mem(1, 0x6c, 2)
cs:  0x22 0x08 0x00 0x00 ...
cs: read_cis_mem(1, 0x76, 2)
cs:  0x22 0x02 0x00 0x00 ...
cs: read_cis_mem(1, 0x7a, 2)
cs:  0x1a 0x07 0x00 0x00 ...
cs: read_cis_mem(1, 0x83, 2)
cs:  0x1b 0x10 0x00 0x00 ...
cs: read_cis_mem(1, 0x95, 2)
cs:  0xff 0xff 0x00 0x00 ...
cs: read_cis_mem(0, 0x0, 5)
cs:  0x00 0x00 0x00 0x00 ...
#ds_ioctl(socket 0, 0xc0206404, 0xbffff934)
ds_ioctl(socket 0, 0xc2946406, 0xbffff934)
```

```
cs: read_cis_mem(1, 0x46, 2)
cs:  0x06 0x00 0xff 0xff ...
ds_ioctl(socket 0, 0xc2946407, 0xbffff934)
ds_ioctl(socket 0, 0xc0206404, 0xbffff934)
ds_ioctl(socket 0, 0xc2946406, 0xbffff934)
ds_ioctl(socket 0, 0xc2946407, 0xbffff934)
ds_ioctl(socket 0, 0xc0206404, 0xbffff934)
ds_ioctl(socket 0, 0xc2946406, 0xbffff934)
cs: read_cis_mem(1, 0x14, 42)
cs:  0x05 0x00 0x41 0x53 ...
ds_ioctl(socket 0, 0xc2946407, 0xbffff934)
ds_ioctl(socket 0, 0xc0506403, 0xbffff8d8)
cardmgr[38]: socket 0: ASUS WL-110 802.11b WLAN CF Card
cardmgr[38]: module /lib/modules/2.4.18-rmk7-pxa1/pcmcia/prism2_cs.o not available
cardmgr[38]: executing: 'modprobe prism2_cs'
init_module: prism2_cs.o: 0.2.0 Loaded
init_module: dev_info is: prism2_cs
ds: register_pccard_driver('prism2_cs')
ds_ioctl(socket 0, 0xc050643c, 0x20137b8)
bind_request(0, 'prism2_cs')
driver=c029c5e0
cs: bind_device(): client 0xc03fe860, sock 0, dev prism2_cs
driver->attach = c4891c30
cs: register_client(): client 0xc03fe860, sock 0, dev prism2_cs
cs: read_cis_mem(1, 0x7c, 7)
cs:  0x03 0x01 0xe0 0x03 ...
cs: read_cis_mem(1, 0x85, 16)
cs:  0xc1 0x01 0x19 0x77 ...
cs: write_cis_mem(1, 0x1f0, 1)
prism2_cs: index 0x01: Vcc 3.3, irq 36, io 0xd2000000-0xd200003f
cs: read_cis_mem(1, 0x1f0, 1)
cs:  0x41 0x60 0xe8 0x3f ...
cs: write_cis_mem(1, 0x1f0, 1)
cs: write_cis_mem(1, 0x1f0, 1)
cs: write_cis_mem(1, 0x1f0, 1)
ds_ioctl(socket 0, 0xc050643d, 0x20137b8)
DS: start get_dev_info
DS: end get_dev_info
ds_ioctl(socket 0, 0xc050643e, 0x2013810)
DS: start get_dev_info
ds_poll(socket 0)
#lsmod
Module                  Size  Used by
prism2_cs              68624   0  (unused)
```

```
p80211                18320   1  [prism2_cs]
ds                     8880   1  [prism2_cs]
i82365.o              9776   1
pcmcia_core           42912   0  [prism2_cs ds i82365.o]
#ifconfig
eth0      Link encap:Ethernet  HWaddr 00:00:C0:FF:EE:08
          inet addr:192.168.1.2  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
          Interrupt:37 Base address:0x300
#cat wlan_ad
#!/bin/sh
wlanctl wlan0 lnxreq_ifstate ifstate=disable
wlanctl wlan0 lnxreq_ifstate ifstate=enable
wlanctl wlan0 dot11req_start ssid=linux-wlan bsstype=independent beaconperiod=100 dtimperiod=3 cfpollable=false cfpollreq=false cfpperiod=3 cfpmaxduration=100 probedelay=100 dschannel=6 basicrate1=2 basicrate2=4 operationalrate1=2 operationalrate2=4 operationalrate3=11 operationalrate4=22
wlanctl wlan0 dot11req_mibset mibattribute=dot11WEPDefaultKeyID=0
wlanctl wlan0 dot11req_mibset mibattribute=dot11ExcludeUnencrypted=true
wlanctl wlan0 dot11req_mibset mibattribute=dot11PrivacyInvoked=true
wlanctl wlan0 dot11req_mibset mibattribute=dot11WEPDefaultKey0=12:34:56:78:90
ifconfig eth0  down
ifconfig wlan0 192.168.0.236 netmask 255.255.0.0 broadcast 192.168.0.255
#./wlan_ad
message=lnxreq_ifstate
  ifstate=disable
  resultcode=success
ident: nic h/w: id=0x800c 1.0.0

ident: pri f/w: id=0x15 1.1.0

ident: sta f/w: id=0x1f 1.4.2

MFI:SUP:role=0x00:id=0x01:var=0x01:b/t=1/1

CFI:SUP:role=0x00:id=0x02:var=0x02:b/t=1/1

PRI:SUP:role=0x00:id=0x03:var=0x01:b/t=4/4

STA:SUP:role=0x00:id=0x04:var=0x01:b/t=1/9
```

```
PRI-CFI:ACT:role=0x01:id=0x02:var=0x02:b/t=1/1

STA-CFI:ACT:role=0x01:id=0x02:var=0x02:b/t=1/1

STA-MFI:ACT:role=0x01:id=0x01:var=0x01:b/t=1/1

Prism2 card SN: 45460871\x00\x00\x00\x00

message=lnxreq_ifstate
  ifstate=enable
  resultcode=success
message=dot11req_start
  ssid='linux-wlan'
  bsstype=independent
  beaconperiod=100
  dtimperiod=3
  cfpperiod=3
  cfpmaxduration=100
  fhdwelltime=no_value
  fhhopset=no_value
  fhhoppattern=no_value
  dschannel=6
  ibssatimwindow=no_value
  probedelay=100
  cfpollable=false
  cfpollreq=false
  basicrate1=2
  basicrate2=4
  basicrate3=no_value
  basicrate4=no_value
  basicrate5=no_value
  basicrate6=no_value
  basicrate7=no_value
  basicrate8=no_value
  operationalrate1=2
  operationalrate2=4
  operationalrate3=11
  operationalrate4=22
  operationalrate5=no_value
  operationalrate6=no_value
  operationalrate7=no_value
  operationalrate8=no_value
  resultcode=success
```

#linkstatus=CONNECTED

也可利用启动脚本在，系统启动时装载这些模块和运行 cardmgr 程序，用 vi 打开 root 根目录下的 mnt/etc/init.d/rcS 文件，添加后内容为：

```
#!/bin/sh
/bin/mount -a
/sbin/insmod /lib/modules/2.4.18-rmk7-pxa1/pcmcia/pcmcia_core.o
/sbin/insmod /lib/modules/2.4.18-rmk7-pxa1/pcmcia/i82365.o
/sbin/insmod /lib/modules/2.4.18-rmk7-pxa1/pcmcia/ds.o
/sbin/insmod /lib/modules/wireless/p80211.o
/sbin/cardmgr
exec /usr/etc/rc.local
```

注意：不能在根目录的 linuxrc 下添加，否则不能正常运行 cardmgr 程序。

5.4 出现问题

1> 网络已经连接，但仍不能 ping 通

原因是必须给网络增加网络密钥(WEP Key)，在 wland_ad 脚本的 ifconfig eth0 down 上面增加如下内容：

```
wlanctl wlan0 dot11req_mibset mibattribute=dot11WEPDefaultKeyID=0
wlanctl wlan0 dot11req_mibset mibattribute=dot11ExcludeUnencrypted=true
wlanctl wlan0 dot11req_mibset mibattribute=dot11PrivacyInvoked=true
wlanctl wlan0 dot11req_mibset mibattribute=dot11WEPDefaultKey0=12:34:56:78:90
```

2> 在/linuxrc 启动脚本中装载时，不能正常运行 cardmgr 程序

修改到/usr/etc/rc.local 脚本中装载

五．CPLD 扩展外部设备

1．扩展 I/O

1．1 I/O 口硬件测试

ADS1.2 下的硬件测试程序：

```
#define CPLD_BaseAddr     (*(volatile U8*)0x08000000)
#define CPLD_BaseAddr_IO  (*(volatile U8*)0x080000c0)
void Test_CPLD_IO(void){
    U8 Temp;
    rBWSCON = rBWSCON & ~(0xf << 8) | (0x0 << 8); //nGCS1=not nUB/nLB(sSBHE),not nWAIT,8-bit
    rBANKCON1 = (0 << 13) | (3 << 11) | (7 << 8) | (1 << 6) | (0 << 4) | (3 << 2) | 0;
    CPLD_BaseAddr_IO = 0x0;
    Delay(10);
    CPLD_BaseAddr_IO = 0xa;
    Delay(10);
```

```
        CPLD_BaseAddr_IO = 0x5;
        Delay(10);
        CPLD_BaseAddr_IO = 0xf;
        Delay(10);
        Temp = CPLD_BaseAddr_IO;
    }
```

1．2 I/O 口 Linux 下驱动

1．2．1 分配虚拟地址空间

在 kernel/include/asm-arm/arch-s3c2410/smdk.h 文件中加入以下宏定义：

```
/* CPLD_7128, nGCS1 */
#define pCSCPLD_BASE         0x08000000
#define vCSCPLD_BASE         0xd0200000
```

在 kernel/arch/arm/mach-s3c2410/smdk.c 文件的下述数组中加入上面定义的相关代码：

```
static struct map_desc smdk_io_desc[] __initdata = {
/* virtual     physical     length       domain      r  w  c  b */
    { vCSCPLD _BASE, pCSCPLD _BASE, 0x00000100, DOMAIN_IO, 0, 1, 0, 0 },//刚增加的
    { vCS8900_BASE, pCS8900_BASE, 0x00100000, DOMAIN_IO, 0, 1, 0, 0 },
    { vCF_MEM_BASE, pCF_MEM_BASE, 0x01000000, DOMAIN_IO, 0, 1, 0, 0 },
    { vCF_IO_BASE, pCF_IO_BASE, 0x01000000, DOMAIN_IO, 0, 1, 0, 0 },
    LAST_DESC
};
```

然后重新编译内核，让内核给 CPLD 芯片分配虚拟地址。

1．2．2 驱动程序

IO_driver.c 清单：

```
#ifndef __KERNEL__
#define __KERNEL__
#endif
#ifndef MODULE
#define MODULE
#endif
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/miscdevice.h>
#include <linux/sched.h>
#include <linux/delay.h>
#include <linux/poll.h>
#include <linux/spinlock.h>
#include <linux/irq.h>
#include <asm/hardware.h>
```

```
#include <asm/io.h>
#define DEVICE_NAME "CPLD_IO"
#define IO_MAJOR 232
#define CPLD_BaseAddr_IO  (*(volatile unsigned char*)(vCSCPLD_BASE + 0xc0))
static ssize_t IO_read(struct file *filp, char *buf, size_t count, loff_t *f_pos){
    unsigned char val;
    val = inb(CPLD_BaseAddr_IO);
    copy_to_user(buf, &val, 1);
    return 1;
}
static ssize_t IO_write(struct file *filp, const char *buf, size_t count, loff_t *f_pos){
    unsigned char val;
    copy_from_user(&val, buf, 1);
    outb(val, CPLD_BaseAddr_IO);
    return 1;
}
static struct file_operations leds_fops={
owner:    THIS_MODULE,
write:    IO_write,
read:     IO_read,
};
static devfs_handle_t devfs_handle;
static int __init IO_init(void){
    int ret;
    ret = register_chrdev(IO_MAJOR, DEVICE_NAME, &leds_fops);
    if (ret < 0){
        printk(DEVICE_NAME "can't register major number");
        return ret;
    }
    devfs_handle = devfs_register(NULL, DEVICE_NAME, DEVFS_FL_DEFAULT, IO_MAJOR, 0, S_IFCHR
| S_IRUSR | S_IWUSR,&leds_fops, NULL);
    printk(DEVICE_NAME ":initialized\n");
    return 0;
}
static void __exit IO_exit(void){
    devfs_unregister(devfs_handle);
    unregister_chrdev(IO_MAJOR, DEVICE_NAME);
}
module_init(IO_init);
module_exit(IO_exit);
```

Makefile 清单：

```
CROSS=/opt/host/armv4l/bin/armv4l-unknown-linux-
INC=/linuette/target/box/kernel/include/
all: IO
```

```
IO:
	$(CROSS)gcc -O2 -DMODULE -D__KERNEL__ -c -I$(INC) IO_driver.c

clean:
	@rm *.o
```

1．2．3 驱动测试程序

IO_test.c 清单：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <fcntl.h>
int main(int argc,char *argv[]){
    int on;
    int IO_no;
    int fd;
    unsigned char val, time = 60;
    fd = open("/dev/PGM_IO", O_RDWR);
    if (fd < 0){
        perror("open device PGM_IO");
        exit(1);
    }
    while (--time){
        if (read(fd, (void *)(&val), 1) < 0){
            perror("read fail");
        }
        printf("read val = %x\n", val & 0x0f);
        val = ~val;
        if (write(fd, (void *)&val, 1) < 0){
            perror("write1 fail");
        }
        sleep(1);
    }
    close(fd);
    return 0;
}
```

Makefile 清单：

```
CROSS=/opt/host/armv4l/bin/armv4l-unknown-linux-
INC=/linuette/target/box/kernel/include/
all: IO_test

IO_test:
```

```
 $(CROSS)gcc IO_test.c -o IO_test

clean:
 rm IO_test
```

1．2．4 出现过的问题

1> 运行测试程序时出现 Oops 错误，如下：

```
#./IO_test
unable to handle kernel paging request at virtual address d02000c0

pgd = c3e38000

*pgd = 00000000, *pmd = 00000000

Internal error: Oops: fffffff

CPU: 0

pc : [<c488017c>]    lr : [<c00530a4>]    Not tainted

sp : c3d3df74  ip : 00000001  fp : c3d3df80

r10: 400dc26c  r9 : c3d3c000  r8 : ffffffe7

r7 : 00000001  r6 : 00000003  r5 : 00000001  r4 : c3d49240

r3 : 00000001  r2 : 0000000f  r1 : c3d49240  r0 : d02000c0

Flags: nZCv  IRQs on  FIQs on  Mode SVC_32  Segment user

Control: C000317F  Table: 33E38000  DAC: 00000015

Process IO_test (pid: 43, stackpage=c3d3d000)

Stack: (0xc3d3df64 to 0xc3d3e000)

df60:          c00530a4 c488017c 60000013 ffffffff c3d3dfa4 c3d3df84 c00530a4

df80: c4880170 00000648 bffffec4 020003c8 00000036 c00177c4 00000000 c3d3dfa8

dfa0: c0017640 c0052eb8 00000648 c001d918 00000003 00000001 00000001 fbad2a84

dfc0: 00000648 bffffec4 020003c8 4001fe94 00000003 0200058c 400dc26c bffffea0
```

```
dfe0: 400984c0 bffffe7c 020006ac 400984c4 60000010 00000003 00000000 00000000

Backtrace:

Function entered at [<c4880160>] from [<c00530a4>]

Function entered at [<c0052ea8>] from [<c0017640>]

 r8 = C00177C4  r7 = 00000036  r6 = 020003C8  r5 = BFFFFEC4

 r4 = 00000648

Code: e1a0c002 e35c0001 e59f003c e3a0200f (e5c02000)

Segmentation fault
```

上述为 Oops 错误，是往无效的地址写入了数据引起，第一行信息表明在申请分配虚拟地址没有成功，不能用这种方法去访问外部地址设备。

下面改成 ioremap 函数申请虚拟地址访问 CPLD 设备。驱动程序其它部分不变，只在初始化函数中增加该函数去申请虚拟地址，在释放函数中再释放该虚拟地址即可，初始化和释放函数修改如下：

```
#define pCPLD_IO_Addr 0x080000c0
static devfs_handle_t devfs_handle;
static int __init IO_init(void){
    int ret;
    ret = register_chrdev(IO_MAJOR,DEVICE_NAME,&leds_fops);
    if (ret < 0){
        printk(DEVICE_NAME "can't register major number");
        return ret;
    }
    devfs_handle = devfs_register(NULL,DEVICE_NAME,DEVFS_FL_DEFAULT,IO_MAJOR,0,S_IFCHR | S_IRUSR | S_IWUSR,&leds_fops,NULL);
    CPLD_BaseAddr_IO= ioremap(pCPLD_IO_Addr, 1);//申请虚拟地址指向 CPLD_IO 设备的物理地址
    printk(DEVICE_NAME ":initialized\n");
    return 0;
}
static void __exit IO_exit(void){
    iounmap(CPLD_BaseAddr_IO);                   //取消虚拟地址的映射
    devfs_unregister(devfs_handle);
    unregister_chrdev(IO_MAJOR,DEVICE_NAME);
}
module_init(IO_init);
module_exit(IO_exit);
```

另外读写 IO 地址的函数 inb()和 outb()也可以分别改成 readb()和 writeb()函数

2> 在测试程序中调用写函数时总是失败，打印出"write1 fail：bad file descriptor"，提示无效的描述符。写函数根本没有执行，在 write 函数之前 printk("<1> IO_write funce\n");没有执行。

原因是测试程序中的 fd = open("/dev/PGM_IO", O_RDWR);简单的写成了 fd = open("/dev/PGM_IO", 0);所以打开的描述符 fd 不具备写的权限。

3> 在驱动程序中执行 printk 函数时，控制台打印不出信息，如：

```
printk("IO_write funce\n");
printk(KERN_INFO "IO_write funce\n");
```

原因：日志级别不够

2．扩展串口 16C554

2．1 硬件测试程序

```
#define        THR       0    //Offset to Transmit hld reg (write),LCR.7 必须为 0
#define        RBR       0    //Receiver holding buffer (read),LCR.7 必须为 0
#define        IER       1    //Interrupt enable register
#define        ISR       2    //Interrupt identification reg
#define        FCR       2    //FIFO Control register
#define        LCR       3    //Line control register
#define        MCR       4    //Modem control register
#define        LSR       5    //Line status register
#define        MSR       6    //Modem status register
#define        DLL       0    //LCR.7 必须为 1
#define        DLM       1    //LCR.7 必须为 1
#define        SR        7    //Scrathpad Register
#define    EXT_COM_COUNT       4                              //扩展的串口数量
#define    EXT_COM_BASE_ADDR  0x08000080                      //扩展串口基地址
#define    rEXTUART_INT_SOURCE (* (volatile unsigned char *)0x080000d8)//串口中断源地址
/*************************************************************************************
函数原形：void Set554Int(unsigned char operation)
功    能：控制 CPLD 使 16C554 的中断屏蔽或输出给 CPU
参    数：operation -- 1 为打开,0 为禁止
*************************************************************************************/
void Set554Int(unsigned char operation){
    unsigned char *pU8;
    pU8 = (unsigned char *)0x80000e0;
    if (operation){
        *pU8 = 0xff;        //打开串口中断
    }
    else{
        *pU8 = 0x00;        //禁止串口中断输出到 CPU
    }
}
/**************************************************************************************
函数原形：void Reset554(void)
```

功　　能：复位16C554和设置bank1的总线宽度及时钟
***/

```
void Reset554(void){
     Set554Int(0);                //控制CPLD,封锁外部串口中断信号输出
     rGPBCON = rGPBCON & (~(0x3 << 12)) | (0x1 << 12);//GPB6 = Output
     rGPBUP &= (~(0x1 << 6));     //GPB6 pull-up enabled
     rGPBDAT |= (0x1 << 6);       //复位脚LCOM_RST = GPB6 = 1
     //   Delay(1);               //Min = 40ns
      rGPBDAT &= (~(0x1 << 6));
      rBWSCON = rBWSCON & ~(0xf << 4) | (0x0 << 4);// not nUB/nLB(sSBHE),not nWAIT,8-bit
      rBANKCON1 = (0 << 13) | (3 << 11) | (7 << 8) | (1 << 6) | (0 << 4) | (3 << 2) | 0;
}
```

1> 测试读写寄存器是否正确

/***
函数原形：void Test_RW_Reg(void)
功　　能：测试读写寄存器是否正确,通过复位后读取寄存器值与datasheet的default值比较
***/

```
volatile unsigned char testval[15];
void Test_RW_Reg(void){
     unsigned char i;
     volatile unsigned char * pBaseAddr = (volatile unsigned char *)0x08000080;
     Reset554();
     for (i=0;i<8;i++){
          Delay(1);
          testval[i] = *(pBaseAddr + i);
     }
    *(pBaseAddr + LCR) = 0x80;
    testval[3] = *(pBaseAddr + LCR);
    testval[8] = *(pBaseAddr+ DLL);
    testval[9] = *(pBaseAddr+ DLM);
    *(pBaseAddr + LCR) = 0xBF;
    testval[3] = *(pBaseAddr + LCR);
    testval[10] = *(pBaseAddr+ 2);
    testval[11] = *(pBaseAddr+ 4);
    testval[12] = *(pBaseAddr+ 5);
    testval[13] = *(pBaseAddr+ 6);
    testval[14] = *(pBaseAddr+ 7);
}
```

2> 测试串口发送接收基本功能

/***
函数原形：void Test_Send_Rec(void)
功　　能：测试串口发送接收基本功能:先连续发送255字节数据,再查寻等待接收数据
***/

```
void Test_Send_Rec(void){
```

```
    volatile unsigned char * pBaseAddr = (volatile unsigned char *)0x08000080;
    unsigned char i,temp;
    *(pBaseAddr + LCR) = 0x80;//写波特率因子寄存器
    *(pBaseAddr + DLL) = 0x30;//波特率:9600=7.3728MHz/16=0x30
    *(pBaseAddr + DLM) = 0x00;
    *(pBaseAddr + LCR) = 0x03;//no parity,stop 1 bit,data 8 bit
    *(pBaseAddr + FCR) = 0x00;//disable FIFO
    *(pBaseAddr + IER) = 0x00;//disable interrupt
    *(pBaseAddr + MCR) = 0x00;
    for (i = 1; i != 0; i++) {    //连续发送(0x1 ~ 0xff)255 个字节
        while (!((*(pBaseAddr+ LSR)) & (0x1 << 5)));//等待 THR empty
        *(pBaseAddr + THR) = i;
    }
    while (1) {
        if ((testval[0]=*(pBaseAddr+ LSR)) & 0x1){//查寻接收
            testval[0] = *(pBaseAddr+ LSR);        //测试接收标志何时清 0
            testval[0] = *(pBaseAddr+ LSR);
            testval[0] = *(pBaseAddr+ LSR);
            temp = *(pBaseAddr+ RBR);
            testval[0] = *(pBaseAddr+ LSR);
            testval[0] = *(pBaseAddr+ LSR);
            testval[0] = *(pBaseAddr+ LSR);
            while (!((*(pBaseAddr+ LSR)) & (0x1 << 5)));//等待 THR empty
            *(pBaseAddr + THR) = temp;
        }
    }
}
```

3> 测试串口中断接收功能

```
/*****************************************************************************************
函数原形：void SerialInitialize(unsigned char comn)
功    能：初始化配置 16C554 寄存器
参    数：comn – 表示第几个串口
*****************************************************************************************/
void SerialInitialize(unsigned char comn){
    unsigned short BaudRate = 0x0030;      //9600
    unsigned char  COMFormat = 0x03;       //no parity,stop 1 bit,data 8 bit
    volatile unsigned char val;
    int i;
    volatile unsigned char * pBaseAddr;
    pBaseAddr=(volatile unsigned char *)(EXT_COM_BASE_ADDR + COMIndex * 0x8);
    *(pBaseAddr+ LCR) = 0x80;              //设置数据格式，并允许写波特率因子寄存器
    *(pBaseAddr+ DLM) = (unsigned char)((BaudRate >> 8) & 0x00ff);//写波特率因子寄存器
    *(pBaseAddr+ DLL) = (unsigned char)(BaudRate & 0x00ff);
    *(pBaseAddr+ LCR)= COMFormat & (~0x80);//禁止写波特率因子寄存器，允许对收发缓冲区操作
```

```
    *(pBaseAddr+ MCR) = 0;                //清除外部输出
    for(i = 0; i < 8; i++){               //清楚状态寄存器的标志位
        val = *(pBaseAddr + i);
    }
    *(pBaseAddr + FCR) = 0;               //disable FIFO
    *(pBaseAddr + IER) = 0;               //禁止所有中断
    *(pBaseAddr + IER) = 0x01;            //允许接收中断
    (*(pBaseAddr+ MCR)) |= 0x08;          //打开串口中断总开关
    if (rINTMSK & BIT_EINT1){             //开串口中断
        ClearPending(BIT_EINT1);
        rINTMSK &= (~BIT_EINT1);
    }
    Set554Int(1);                         //开外部串口中断锁
}
/*************************************************************************************
函数原形：void __irq ExtUart_Int(void)
功    能：16C554 串口中断,先读中断源，判断是哪个串口中断，作相应处理
*************************************************************************************/
void __irq ExtUart_Int(void){
     unsigned char Port = rEXTUART_INT_SOURCE;
     unsigned char lsr,val=0,ch,reg;
     int i;
     volatile unsigned char * pBaseAddr;
     if (Port == 0){
         Uart_Printf("Com interrupt error, no interrupt detect.\n");
     }
     else{
          for (i = 0; i < 8; i++){
               if ((Port >> i) & 0x01){
                   pBaseAddr = (volatile unsigned char *)(EXT_COM_BASE_ADDR + i * 0x8);
                   lsr = *(pBaseAddr + LSR);//let me read it first,to clear it's interrupt
      state
                   val = *(pBaseAddr + ISR);//获取中断状态
                   if (val & 0x01){          //无中断则退出
                       Uart_Printf("Com interrupt error, no interrupt detect.\n");
                   }
                   else if (val & 0x04){
                       Uart_Printf("Receive interrupt\n");
                       while ((*(pBaseAddr + LSR)) & 0x1){//Receive data ready.
                           testval[i] = *(pBaseAddr + RBR);//从 16C554 读取接收到的数据
                           while (!((*(pBaseAddr + LSR)) & (0x1 << 5)));//等待 THR empty
                           *(pBaseAddr + THR) = testval[i++];//返回接收到的数据
                       }
                   }
```

```
                else if (val & 0x02){
                    Uart_Printf("Sender interrupt\n");
                }
            }
        }
    }
    ClearPending(BIT_EINT1);
}
```

4> 测试 FIFO 接收功能

只要将上述 SerialInitialize 函数中的

```
*(pBaseAddr + FCR) = 0;              //disable FIFO
```

修改成：

```
*(pBaseAddr + FCR) = 0x87;           //允许 FIFO 接收 8 字节触发，同时清除接收和发送 FIFO
```

注：在调试 FIFO 时，当只发送一个数据而没有发送设定的 RX FIFO trigger level 数据时也会产生中断，原因接收数据超时而产生的是 Receive Data time-out 中断(ISR = 0xcc)，所以在接收中断中要判断 LSR[0](Recive data ready)是否为 1，如果为 1 则 RHR 里的数据有效，当读完 RHR 里的数据后 LSR[0]会自动清 0。

5> 流控测试

SC16C554 的 datasheet 描述的自动流控设置寄存器 EFR 不能读写，尽管将 FCR=0xBF 再对其地址 0x02 进行访问，但访问的仍为 FCR=0 时的 ISR 寄存器。所以只能手动设置 MCR[1]将/RTS 脚置高或拉低。如下：

```
testval[0] = *(pBaseAddr + MCR);
*(pBaseAddr + MCR) = testval[0] | 0x2;// /RTS = 0
testval[0] = *(pBaseAddr + MCR);
*(pBaseAddr + MCR) = testval[0] & (~0x02);// /RTS = 1
```

2．2 linux 驱动

2．2．1 说明

1> 能够设置读取各参数（如：波特率、数据位、停止位、奇偶校验方式、流控等），用 ioctl 实现。

2> 16C554 扩展的四个串口共用一个中断源，一个驱动。编译成模块装载后，在/dev 目录下创建 4 个不同子设备号的驱动文件，在驱动程序中利用不同的子设备号进行区分。

2．2．2 驱动源程序

```
#ifndef __KERNEL__
#define __KERNEL__
#endif
#ifndef MODULE
#define MODULE
#endif
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/miscdevice.h>
#include <linux/sched.h>
```

```
#include <linux/delay.h>
#include <linux/poll.h>
#include <linux/spinlock.h>
#include <linux/interrupt.h>
#include <asm/hardware.h>
#include <asm/io.h>
#include <asm/arch/cpu_s3c2410.h>
#include <asm/irq.h>
#include <asm-arm/mach/irq.h>
#include <linux/time.h>
#define    DEVICE_NAME          "PGM_EXTCOM"
#define    uart_MAJOR           235
#define    BUFSIZE              128
#define    TIMEOUT              10

#define    pEXT_COM0_BASE_ADDR   0x08000080
#define    pEXT_COM_INTS_ADDR    0x080000d8//interrupt source
#define    pEXT_COM_INTCTL_ADDR  0x080000e0//0xff enable CLPD uart interrupts out, 0x00
disable
#define    EXT_COM_RST          GPIO_B6
#define    EXT_COM_INT          IRQ_EINT1
#define    EXT_COM_NUM          4
#define    THR                  0//Offset to Transmit hld reg (write),LCR.7 is 0
#define    RBR                  0//Receiver holding buffer (read),LCR.7 is 0
#define    IER                  1//Interrupt enable register
#define    ISR                  2//Interrupt identification reg
#define    FCR                  2//FIFO Control register
#define    LCR                  3//Line control register
#define    MCR                  4//Modem control register
#define    LSR                  5//Line status register
#define    MSR                  6//Modem status register
#define    SR                   7//Scrathpad Register
#define    DLL                  0//LCR.7 is 1
#define    DLM                  1//LCR.7 is 1
#define    BaudRate_div         (7372800/16)
#define    EXT_COM_INTBit       (1 << 12)
unsigned char  EXT_COM_USE[4] = {0, 0, 0, 0};
unsigned int   EXT_COM_PARAM[4] = {0x703, 0x703, 0x703, 0x703};
//receve interrupt disable,9600,no flow,no parity,stop 1bit,,8bit
static const unsigned int BaudRateTab[16] = {
    0x105d,//110
    0x0900,//200
    0x0600,//300
    0x0300,//600
```

```
        0x0180,//1200
        0x00c0,//2400
        0x0060,//4800
        0x0030,//9600
        0x0020,//14400
        0x0018,//19200
        0x000c,//38400
        0x0008,//56000
        0x0008,//57600
        0x0004,//115200
        0x0002,//230400
        0x0001,//460800
    };
    static const unsigned int pEXT_COM_BASE_ADDR[] = {0x08000080, 0x08000088, 0x08000090,
0x08000098};
    void *vEXT_COM_BASE_ADDR[] = {0x00000000, 0x00000000, 0x00000000, 0x00000000};
    void *vEXT_COM_INTS_ADDR;
    void *vEXT_COM_INTCTL_ADDR;

    static DECLARE_WAIT_QUEUE_HEAD(rx_queue0);
    static DECLARE_WAIT_QUEUE_HEAD(rx_queue1);
    static DECLARE_WAIT_QUEUE_HEAD(rx_queue2);
    static DECLARE_WAIT_QUEUE_HEAD(rx_queue3);
    void *rx_queue[EXT_COM_NUM];
    struct Queue{
        char * buf;
        char * rear;
        char * front;
        unsigned int num;
        unsigned char emp;//1:empty, 0:no empty
        unsigned char full;//1:full, 0:no full
        unsigned char usr;//1:usr, 0:no usr,usr interrupts
    }COM0_rx_buf, COM1_rx_buf, COM2_rx_buf, COM3_rx_buf;
    struct Queue *COM_rx_buf[EXT_COM_NUM];
    static int Init_Queue(struct Queue *pQueue){
        printk("<1> Init_Queue %x\n", pQueue);
        pQueue->emp = 1;
        pQueue->full = 0;
        pQueue->num = BUFSIZE;
        pQueue->buf = kmalloc(sizeof(pQueue->buf) * pQueue->num, GFP_KERNEL);
        if (!pQueue->buf){
            printk("<1> kmalloc faill\n");
            return -ENOMEM;
        }
```

```
    else{
        pQueue->front = pQueue->rear = pQueue->buf + pQueue->num - 1;//point last
        printk("<1> pQueue->rear = %x\n", pQueue->rear);
    }
    return 0;
}
void Reset554(void){
    set_gpio_ctrl(EXT_COM_RST | GPIO_PULLUP_EN | GPIO_MODE_OUT);
    write_gpio_bit(EXT_COM_RST, 1);
    udelay(10);
    write_gpio_bit(EXT_COM_RST, 0);
}
void SerialInit(const void * vEXT_COM_BASE_ADDR, const unsigned int PARAM){
    unsigned short BaudRate = BaudRateTab[(PARAM >> 8) & 0xf];//9600
    unsigned char  COMFormat = PARAM & 0x3f;//no parity,stop 1 bit,data 8 bit
    volatile unsigned char val;
    unsigned char i;
    writeb(0x80, vEXT_COM_BASE_ADDR + LCR);//set baudrate
    writeb((BaudRate >> 8) & 0x00ff, vEXT_COM_BASE_ADDR + DLM);
    writeb(BaudRate & 0x00ff, vEXT_COM_BASE_ADDR + DLL);
    writeb(COMFormat & (~0x80), vEXT_COM_BASE_ADDR + LCR);
    wmb();
    for (i = 0; i < 8; i++){
        readb(vEXT_COM_BASE_ADDR + i);  //flash state register flags
        rmb();
    }
    writeb(0xc7, vEXT_COM_BASE_ADDR + FCR);//enable FIFO, RX FIFO trigger level is 14 byte,
FIFO reset
    writeb(0x01, vEXT_COM_BASE_ADDR + IER);//only enable rx interrupt
    if (PARAM & EXT_COM_INTBit){
        writeb(0x08, vEXT_COM_BASE_ADDR + MCR);//eable interrupts
    }
    else{
        writeb(0x0, vEXT_COM_BASE_ADDR + MCR);//disable interrupts
    }
    wmb();
}
static void uart_irq_handle(int irq, void *dev_id, struct pt_regs *regs){
    volatile unsigned char lsr,isr;
    unsigned char port = readb(vEXT_COM_INTS_ADDR);
    unsigned char n;
    struct Queue *qCOM_rx_buf;
    int inc = 0;
    for (n = 0; n < EXT_COM_NUM; n++){
```

```
            if (!EXT_COM_USE[n]) continue;
            if (((port >> n) & 0x01) && (qCOM_rx_buf = COM_rx_buf[n])){
                printk("<1> COM_rx_buf[%d] = %x\n", n, COM_rx_buf[n]);
                isr = readb(vEXT_COM_BASE_ADDR[n] + ISR);
                if ((!(isr & 0x01)) && (isr & 0x04)){
                    while ((!qCOM_rx_buf->full) && ((lsr = readb(vEXT_COM_BASE_ADDR[n] + LSR))
& 0x01)) {               //recive data ready, read later is clear
                        if(++(qCOM_rx_buf->rear) == (qCOM_rx_buf->buf+qCOM_rx_buf->num)){
                            qCOM_rx_buf->rear = qCOM_rx_buf->buf;
                        }
                        if (qCOM_rx_buf->rear == qCOM_rx_buf->front){
                            qCOM_rx_buf->full = 1;
                        }
                        *qCOM_rx_buf->rear = readb(vEXT_COM_BASE_ADDR[n] + RBR);
                        rmb();
                        inc++;
                    }
                    if (inc){
                       qCOM_rx_buf->emp = 0;
                        wake_up_interruptible(rx_queue[n]);
                    }
                }
            }
        }
    }
    static ssize_t uart_read(struct file *filp, char *buf, size_t count, loff_t *f_pos){
        unsigned long flag;
        char rx_buf[BUFSIZE];
        size_t rx_buf_count = 0;
        unsigned int i;
        struct inode *inode = filp->private_data;
        struct Queue *qCOM_rx_buf = COM_rx_buf[MINOR(inode->i_rdev)];
        printk("uart_read  MINOR  is  %d,  filp->f_count  is  %d\n",  MINOR(inode->i_rdev),
filp->f_count);
        local_irq_save(flag);
        while (rx_buf_count < count){
            if (!qCOM_rx_buf->emp){
                if (++qCOM_rx_buf->front == (qCOM_rx_buf->buf + qCOM_rx_buf->num)) {
                    qCOM_rx_buf->front = qCOM_rx_buf->buf;
                }
                if (qCOM_rx_buf->front == qCOM_rx_buf->rear){
                   qCOM_rx_buf->emp = 1;//is empty
                }
                rx_buf[rx_buf_count++] = *qCOM_rx_buf->front;
```

```
                qCOM_rx_buf->full = 0;
            }
            else{
                interruptible_sleep_on(rx_queue[MINOR(inode->i_rdev)]);
    //                    printk("<1>out sleep\n");
            }
        }
        copy_to_user(buf, rx_buf, rx_buf_count);
        local_irq_restore(flag);
        return rx_buf_count;
    }
    static ssize_t uart_write(struct file *filp, const char *buf, size_t count, loff_t *f_pos){
        int i;
        char tx_buf[BUFSIZE];
        unsigned char lsr;
        size_t tx_buf_count = 0;
        struct inode *inode = filp->private_data;
        printk("<1> uart_write: minor=%d, filp->f_count is %d\n", MINOR(inode->i_rdev),
filp->f_count);
        copy_from_user(tx_buf, buf, count);
        for (i = 0; i < count; i++){
            while (!(lsr = readb(vEXT_COM_BASE_ADDR[MINOR(inode->i_rdev)] + LSR) & (0x1 <<
5)));//wait THR empty
            writeb(tx_buf[i], vEXT_COM_BASE_ADDR[MINOR(inode->i_rdev)] + THR);
        }
        return count;
    }
    static int uart_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long
arg){
        printk("<1> uart_ioctl MINOR is %d, filp->f_count is %d\n", MINOR(inode->i_rdev),
filp->f_count);
        switch (cmd & 0x0f){
            case 1://read
                    return EXT_COM_PARAM[MINOR(inode->i_rdev)];
            case 2://write
                    EXT_COM_PARAM[MINOR(inode->i_rdev)] = arg & 0x1fff;
                    disable_irq(EXT_COM_INT);
                    printk("%d write param is %x\n", MINOR(inode->i_rdev), arg);
                    SerialInit(vEXT_COM_BASE_ADDR[MINOR(inode->i_rdev)],
EXT_COM_PARAM[MINOR(inode->i_rdev)]);
                    enable_irq(EXT_COM_INT);
                    break;
            default:
                    return -EINVAL;
```

```
        }
        return 0;
    }
    static int uart_open(struct inode *inode, struct file *filp){
        printk("<1> uart_open MINOR is %d\n", MINOR(inode->i_rdev));
        printk("<1> filp is %x, filp->count is %d\n", filp, filp->f_count);
        MOD_INC_USE_COUNT;
        Init_Queue(COM_rx_buf[MINOR(inode->i_rdev)]);
        EXT_COM_USE[MINOR(inode->i_rdev)] = 1;//usr
        filp->private_data = inode;//use read and write function
        EXT_COM_PARAM[MINOR(inode->i_rdev)] |= EXT_COM_INTBit;//enable interrupts
        SerialInit(vEXT_COM_BASE_ADDR[MINOR(inode->i_rdev)],
EXT_COM_PARAM[MINOR(inode->i_rdev)]);
        return 0;
    }
    static int uart_release(struct inode *inode, struct file *filp) {
        printk("<1> uart_release MINOR is %d, file->f_counnt is %d\n", MINOR(inode->i_rdev),
filp->f_count);
        MOD_DEC_USE_COUNT;
        kfree(COM_rx_buf[MINOR(inode->i_rdev)]->buf);
        EXT_COM_USE[MINOR(inode->i_rdev)] = 0;//no usr
        EXT_COM_PARAM[MINOR(inode->i_rdev)] &= EXT_COM_INTBit;//disable interrupts
        SerialInit(vEXT_COM_BASE_ADDR[MINOR(inode->i_rdev)],
EXT_COM_PARAM[MINOR(inode->i_rdev)]);
        return 0;
    }
    static int uart_flush(struct file *filp){
        unsigned int i;
        struct inode *inode = filp->private_data;
        printk("<1> uart_flush MINOR is %d, filp->f_count is %d\n", MINOR(inode->i_rdev),
filp->f_count);
        return 0;
    }
    static struct file_operations uart_fops={
    owner:    THIS_MODULE,
    open:uart_open,
    release:  uart_release,
    flush:    uart_flush,
    write:    uart_write,
    read:     uart_read,
    ioctl:    uart_ioctl,
    };
    static devfs_handle_t devfs_uart[4], devfs_uart_dir;
    static int __init uart_init(void){
```

```
        int ret;
        unsigned char i;
        ret = register_chrdev(uart_MAJOR, DEVICE_NAME, &uart_fops);
        if (ret < 0){
            printk(DEVICE_NAME "cant't register major num\n");
            return ret;
        }
        for (i = 0; i < EXT_COM_NUM; i++){
            vEXT_COM_BASE_ADDR[i] = ioremap(pEXT_COM_BASE_ADDR[i], 16);
            printk("<1> vEXT_COM_BASE_ADDR[%d]=%x\n", i, vEXT_COM_BASE_ADDR[i]);
        }
        vEXT_COM_INTS_ADDR = ioremap(pEXT_COM_INTS_ADDR, 1);
        vEXT_COM_INTCTL_ADDR = ioremap(pEXT_COM_INTCTL_ADDR, 1);
        writeb(0x0, vEXT_COM_INTCTL_ADDR);//disable COM INT out
        BWSCON = BWSCON & ~(0xf << 4) | (0x0 << 4);//nGCS1=not nUB/nLB(sSBHE),not nWAIT,8-bit
        BANKCON1 = (0 << 13) | (3 << 11) | (7 << 8) | (1 << 6) | (0 << 4) | (3 << 2) | 0;
        Reset554();
        for (i = 0; i < EXT_COM_NUM; i++){
            SerialInit(vEXT_COM_BASE_ADDR[i], EXT_COM_PARAM[i]);
            EXT_COM_USE[i] = 0;
        }
        set_external_irq(EXT_COM_INT, EXT_HIGHLEVEL, GPIO_PULLUP_DIS);
        request_irq(EXT_COM_INT,   uart_irq_handle,   SA_INTERRUPT,   DEVICE_NAME"IRQ_EXTCOM",
NULL);
        writeb(0xff, vEXT_COM_INTCTL_ADDR);//enable COM INT out
        COM_rx_buf[0] = &COM0_rx_buf;
        COM_rx_buf[1] = &COM1_rx_buf;
        COM_rx_buf[2] = &COM2_rx_buf;
        COM_rx_buf[3] = &COM3_rx_buf;
        rx_queue[0] = &rx_queue0;
        rx_queue[1] = &rx_queue1;
        rx_queue[2] = &rx_queue2;
        rx_queue[3] = &rx_queue3;
        printk(KERN_INFO DEVICE_NAME ": init OK\n");
        return(0);
    }

    static void __exit uart_exit(void){
     unsigned char i;
        for (i = 0; i < EXT_COM_NUM; i++){
            iounmap(vEXT_COM_BASE_ADDR[i]);
        }
        iounmap(vEXT_COM_INTS_ADDR);
        iounmap(vEXT_COM_INTCTL_ADDR);
```

```
    free_irq(EXT_COM_INT, NULL);
    unregister_chrdev(uart_MAJOR, DEVICE_NAME);
}
module_init(uart_init);
module_exit(uart_exit);
MODULE_LICENSE("GPL");
```

2．2．3 测试程序

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#define  CSIZE   0x003//data bit
#define  CS5      0x000
#define  CS6     0x001
#define  CS7     0x002
#define  CS8     0x003
#define  CSTOPB  0x004//stop bit
#define  CSTOP1  0x000
#define  CSTOP2  0x001
#define  CPAR    0x038//parity
#define  PARNO   0x000
#define  PARODD  0x008
#define  PARENB  0x018
#define  PARMAK  0x028
#define  PARSPE  0x038
#define  CFCTL   0x0c0//flow control
#define  CFCTLN  0x000
#define  CFCTLH  0x040
#define  CFCTLS  0x080
#define  CBAUD   0xf00//baudrate
#define  B110    0x000
#define  B200    0x100
#define  B300    0x200
#define  B600    0x300
#define  B1200   0x400
#define  B2400   0x500
#define  B4800   0x600
#define  B9600   0x700
#define  B14400  0x800
#define  B19200  0x900
#define  B38400  0xa00
#define  B56000  0xb00
#define  B57600  0xc00
```

```
#define  B115200 0xd00
#define  B230400 0xe00
#define  B460800 0xf00
#define  INTEN   0x1000//must be enable
#define BufLen  128
char SendBuffer[BufLen];
char RecBuffer[BufLen];
int WriteByte, ReadByte;
void EXT_COM_test(int fd){
     int WriteByte, ReadByte;
     char RecBuffer[BufLen];
     WriteByte = write(fd, SendBuffer, BufLen);
     printf("fd %d write %d byte\n", fd, WriteByte);
     ReadByte = read(fd, RecBuffer, 20);
     printf("fd %d read %d byte\n", fd, ReadByte);
     WriteByte = write(fd, RecBuffer, ReadByte);
     printf("fd %d write %d byte\n", fd, WriteByte);
     close(fd);
}
int main(int argc, char *argv[]){
 const char *COM_DIR[] = {"/dev/PGM_EXTCOM0", "/dev/PGM_EXTCOM1", "/dev/PGM_EXTCOM2",
"/dev/PGM_EXTCOM3"};
 unsigned int COM_PARAM[4] = {CS8 | CSTOP1 | PARMAK | CFCTLN | B115200 | INTEN,
                              CS8 | CSTOP1 | PARMAK | CFCTLN | B115200 | INTEN,
                              CS8 | CSTOP1 | PARMAK | CFCTLN | B115200 | INTEN,
                              CS8 | CSTOP1 | PARMAK | CFCTLN | B115200 | INTEN};
    int COM_fd[4];
    unsigned int i;
    pid_t pid;
    for (i = 0; i < BufLen; i++){
        SendBuffer[i] = i;
//        printf("SendBuffer[%d]=%x\n",SendBuffer[i]);
    }
//    fd = open(argv[1], O_RDWR);
    for (i = 0; i < 4; i++){
        COM_fd[i] = open(COM_DIR[i], O_RDWR);
        if (COM_fd[i] < 0){
            printf("open %s fail!\n", COM_DIR[i]);
        }
        else{
            printf("%s fd is %d\n", COM_DIR[i], COM_fd[i]);
            ioctl(COM_fd[i], (i << 4) | 2, COM_PARAM[i] & 0x1fff);
            if ((pid = fork()) < 0){
                printf("fork err\n");
```

```
            }
            else if (pid == 0){
                EXT_COM_test(COM_fd[i]);
                exit(0);
            }
            else{
                close(COM_fd[i]);
            }
        }
    }
    wait(NULL);
    return 0;
}
```

2．2．4 出现过的问题

1> 刚装载驱动程序时就出现 oops 错误，提示对 0x00000001 无效的指针访问。

原因是串口中断函数中对指针进行操作，而该指针是在 open 函数中初始化的，但因串口在装载时就初始化（以后因放在 open 中初始化）好了中断，在装载后又收到了串口数据而产生串口中断。所以在执行中断函数过程中对未初始化的指针进行了操作。

2> 当发送多个数据时，每次都只有第一个数正确，后面都错。

原因是发送和接收奇偶校验方法不一样。当出错后，LSR=0xE9(FIFO data err = 1;Framing error = 1)。而只发第一个数据时，此时 LSR=0x61(表明没有错误)。结果发现是在 LCR 设置成 0x2b(force parity “1”)，修改发送方的 0x03(no parity)就可以。

3> 创建多个子设备问题

起初分别用 4 个不同子设备号调用四次 devfs_register（设备文件系统）函数创建时，只创建了第一个，其后的几个都出现“devfs_register(EXTCOM)：could not append to parent，err：-17”。后来只能用 register_chrdev 函数注册，再分别由 mknod /dev/EXT_COM0 c major minor 方式创建 4 个不同子设备号（minor）的驱动文件。

4> 各函数中区分子设备的方法

Open、release 和 ioctl 函数中因由 inode 参数，可以直接用 MINOR(inode->i_rdev)得到子设备号来区分；read 和 write 没有 inode 参数，只有 struct file *filp 参数，只能利用 filp->private_data 进行传递。方法是在 open 函数中执行 filp->private_data = inode;在 read 函数中执行 struct inode *inode = filp->private_data;再利用 MINOR(inode->i_rdev)得到子设备号。

5> 如果反复的给没有打开的子设备发数据产生中断时，系统会打印出下述信息后死掉！

IRQ LOCK：IRQ1 is locking the system, disabled

目前的解决方法只能是设置 16C554 的 MCR 寄存器将没有打开的子设备中断禁止。在 open 函数中将打开的使能，在 release 中将其禁止。

6> close 系统调用不执行驱动中的 release 函数！即在测试程序中分别打开(open)4 个子设备，然后创建(fork)4 个子进程分别调用 read(阻塞型)接收 4 个串口数据，接收完后再关闭(close)对应子设备的文件描述符后返回父进程。但在子进程中执行 close 时，并没有执行驱动程序中的 release 函数，而是 4 个子进行都执行了 close 后才一下子执行了 4 次 release。

原因是应用程序上 fork 了子进程，而子进程继承了父的资源，当然也包括了在父进程打开的设备驱动程序的文件描述符，所以此时两个进程都拥有了该设备，因此要在父进程和子进程中都执行 close()后才能使 filp->f_count 减为 0，从而执行 release 函数，所以可以在 fork 子进程成功后，父进行执行一次 close()，这样就使当子进程执行 close 时马上执行 release。上面提到的当 4 个设备都 close 后，release

执行了 4 次。是因为父进程结束了，虽然没有执行 close，但随着进程的退出，所有资源都将自动释放，也就包括刚打开的 4 个设备驱动描述符

在《Linux 设备驱动 Edition 2》的 release 相关部分是这样介绍："并不是每个 close 系统调用都会引起对 release 方法的调用，仅仅是那些真正释放设备数据结构的 close 调用才会调用这个方法，因此名字是 release 而不是 close。内核维护一个 file 结构被使用多少次的计数器。无论是 fork 还是 dup 都不创建新的数据结构（仅有 open 创建），它们只是增加已有结构中的计数。只有在 file 结构的计数归 0 时，close 系统调用才会执行 release 方法，这只在删除这个结构时才会发生……"。上面讲的计数器是 file 结构中的 f_count，驱动中在每个函数将 filp->f_count 打印观察发现每进入函数一次就会使其加 1，退出后再减 1。当调用子进程的 close 时打印该值还是 2(可在驱动中增加 flush 函数，只要执行 open 系统调用，它就会执行)，这正是由于父进种 open 一次加 1，fork 以后又加 1，所以子进程 close 时比原先还是多了 1。

六．PWM 驱动蜂鸣器

1．驱动源码

```
#ifndef __KERNEL__
#define __KERNEL__
#endif
#ifndef MODULE
#define MODULE
#endif
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/miscdevice.h>
#include <linux/sched.h>
#include <linux/delay.h>
#include <linux/poll.h>
#include <linux/spinlock.h>
//#include <linux/irq.h>
#include <linux/interrupt.h>
#include <asm/hardware.h>
#include <asm/io.h>
#include <asm/arch/cpu_s3c2410.h>
#include <asm/irq.h>
#include <asm-arm/mach/irq.h>
#include <linux/time.h>

#define DEVICE_NAME "PGM_BUZZ"
#define BUZZ_MAJOR 233

static unsigned int PWM2_Fre = 10000;//10KHz
MODULE_PARM(PWM2_Fre, "i");
```

```
static unsigned int RunCount;
static unsigned int usage;
static void pwm_irq_handle(int irq, void *dev_id, struct pt_regs *regs){
    if ((RunCount > 0)){
        if (RunCount < 0xffffffff){
            RunCount--;
        }
    }
    else {
        TCON = TCON & (~(0xf << 12)) | (0xa << 12);//Timer2: stop
    }
}

static int buzz_open(struct inode *inode, struct file *filp){
    int ret;
    unsigned long flag;
    if (usage == 0){
        local_irq_save(flag);
//          printk("<1> TCFG0 = %x, TCFG1 = %x\n", TCFG0, TCFG1);
        TCFG1 = TCFG1 & (~(0xf << 8)) | (0 << 8);   //Timer2 MUX = 1/2
        TCNTB2=(s3c2410_get_bus_clk(GET_PCLK))/(((TCFG0>>8)&0xff)+1)/2/ PWM2_Fre;
        TCMPB2=TCNTB2/2;              //50%
        TCON = TCON & (~(0xf << 12)) | (0xa << 12);//Timer2:auto reload;Inverter off;Update
TCNTB0,TCMPB0;stop
//          printk("<1> TCNTB2 = %x, TCMPB2 = %x, TCON = %x\n", TCNTB2, TCMPB2, TCON);
        RunCount = 0;
        local_irq_restore(flag);
    }
    usage++;
    MOD_INC_USE_COUNT;
    return 0;          // success
}

static int buzz_release(struct inode *inode, struct file *filp){
    unsigned long flag;
    MOD_DEC_USE_COUNT;
    usage--;
    return 0;
}

static int buzz_ioctl(struct inode *inode, struct file *filp,unsigned int cmd, unsigned long
arg){
    unsigned long flag;
```

```
    switch(cmd) {
        case 0:                          //stop
                RunCount = 0;
                break;
        case 1:                          //start
                local_irq_save(flag);
                if (arg > 0) {
                    RunCount = arg;
                    TCON = TCON & (~(0xf << 12)) | (0x9 << 12);//Timer2: start
                }
                local_irq_restore(flag);
                break;
        default:
                break;
    }
    return 0;
}

static struct file_operations  buzz_fops={
 owner:   THIS_MODULE,
 ioctl:   buzz_ioctl,
 open:    buzz_open,
 release: buzz_release,
};
static devfs_handle_t devfs_handle;
static int __init pwm_init(void){
//    printk("<1> INTMSK = %x\n", INTMSK);
    request_irq(IRQ_TIMER2, pwm_irq_handle, SA_INTERRUPT, DEVICE_NAME"IRQ_TIMER2", NULL);
//    printk("<1> INTMSK = %x\n", INTMSK);
    devfs_handle = devfs_register(NULL, DEVICE_NAME, DEVFS_FL_DEFAULT, BUZZ_MAJOR, 0,
S_IFCHR | S_IRUSR | S_IWUSR, &buzz_fops, NULL);
    set_gpio_ctrl(GPIO_B2 | GPIO_PULLUP_EN | GPIO_MODE_TOUT);//TOUT2
    printk(KERN_INFO DEVICE_NAME ": init OK\n");
    return(0);
}

static void __exit pwm_cleanup(void){
    free_irq(IRQ_TIMER2, NULL);
    devfs_unregister(devfs_handle);
}

module_init(pwm_init);
module_exit(pwm_cleanup);
MODULE_LICENSE("GPL");
```

2．驱动测试程序

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <fcntl.h>
int main(int argc,char *argv[]){
    unsigned int cmd;
    unsigned int PWM_Counter;
    int fd;
    int ret;
    if    ((argc    !=    3)    ||    (sscanf(argv[1],"%x",&cmd)    !=    1)    ||
(sscanf(argv[2],"%x",&PWM_Counter) != 1) || (cmd > 1)){
        fprintf(stderr,"Usage: BUZZ_test 1|0 Counter\n");
        exit(1);
    }
    fd = open("/dev/PGM_BUZZ",0);
    if (fd < 0) {
       perror("open device PGM_BUZZ");
       exit(1);
    }
    if ((ret = ioctl(fd, cmd, PWM_Counter)) < 0){
        perror("ioctl");
    }
    close(fd);
    return 0;
}
```

3．出现过的问题

1> **Proc/interrupts 中没有申请的中断项**，即#cat /proc/interrupts 是没有出现 PGM_BUZZ IRQ_TIMER2 一项。

原因是起初将申请中断和释放中断分别放在 open 和 release 函数中，而在测试程序中 open 再 ioctl 后马上就 close 了，所以当执行 cat 命令时中断早已经释放。

起初还以为设置定时中断时除了调用 request()外还需要其它操作，因为在设置外部中断时除调用 request()外还调用了 set_external_irq(),set_external_irq 设置相应 GPIO 口为外部中断功能和使能上拉等。另外，内核将 TIMER4 作为内核调度时钟，它的初始化程序如下：

```
 TCON = (TCON_4_AUTO | TCON_4_UPDATE | COUNT_4_OFF);
 timer_irq.handler = s3c2410_timer_interrupt;//中断处理程序
//    printk("<1> INTMSK = %x\n", INTMSK);//自己加的，用于观察中断屏蔽寄存器的值
 setup_arm_irq(IRQ_TIMER4, &timer_irq);
//    printk("<1> INTMSK = %x\n", INTMSK);
```

```
TCON = (TCON_4_AUTO | COUNT_4_ON);
```

在调用 setup_arm_irq()函数的前后打印出的 INTMSK 值(可在系统启动信息中找到)分别为 fffffff 和 ffffbfff，表明在setup_arm_irq()函数中清0了TIMER4的中断屏蔽位，但它同样没有调用request()函数。所以我也就用这种方法去实现TIMER2的中断申请，但还是没有成功！后来查书发现request()其实也是调用 setup_arm_irqd()函数的。它给中断描述结构申请空间，并初始化该结构，然后调用setup_arm_irq函数把中断加入到全局中断描述结构数组中。源码如下：

```
int request_irq(unsigned int irq, void (*handler)(int, void *, struct pt_regs *),
       unsigned long irq_flags, const char * devname, void *dev_id){
 unsigned long retval;
 struct irqaction *action;
 if (irq >= NR_IRQS || !irq_desc[irq].valid || !handler ||
     (irq_flags & SA_SHIRQ && !dev_id))
      return -EINVAL;
 action = (struct irqaction *)kmalloc(sizeof(struct irqaction), GFP_KERNEL);
 if (!action)
      return -ENOMEM;
 action->handler = handler;
 action->flags = irq_flags;
 action->mask = 0;
 action->name = devname;
 action->next = NULL;
 action->dev_id = dev_id;
 retval = setup_arm_irq(irq, action);
 if (retval)
      kfree(action);
 return retval;
}
```

后来重新使用request()函数申请中断，且在前后也加入了打印INTMSK值观察，发现TIMER2的屏蔽位有清0。后来才发现是release函数中释放了中断。

2> 没有产生中断，#cat /proc/interrupts时PGM_BUZZ IRQ_TIMER2的中断次数一直为0

原因是TCON = TCON & (~(0xf << 12)) | (0x9 << 12);设成TCON = TCON & (~(0xf << 12)) | (0xb << 12); 即将Timer 2 manual updata位置1，使其永远都工作在Update TCNTB2 & TCMPB2

七．485网络驱动

1．硬件测试

1．1 测试代码清单

```
#include <stdio.h>
#include <stdlib.h>
#include "..\Inc\2410addr.h"
#include "..\Inc\2410lib.h"
#include "..\Inc\option.h"
```

```
void Send00485(void);
void Sendbyte485(unsigned char data);
void Uart1_Init(int mclk,int baud);
/******************************************************************************
函数原形：void Uart1_Init(int pclk,int baud)
功能描述：初始化 UART1(485)
参    数：
******************************************************************************/
void Uart1_Init(int pclk,int baud){
    int i;
    if (pclk == 0)
    pclk = PCLK;
    rUFCON1 = 0x0; //FIFO disable
    rUMCON1 = 0x0; //AFC disable
    rULCON1 = 0x33;//8 位数据，1 位停止位，第 9 位(奇偶位)强制为 1
    rUCON1 = 0x245;//rx=edge,tx=level,disable timeout int.,,normal,interrupt or polling
    rUBRDIV1 = ( (int)(pclk/16./baud) -1 );//bps = 38400
    for (i = 0; i < 100;i++);
}
/*********************************************************************************
函数原形：void Output_Relay485(unsigned char FunNum,unsigned char NetID,unsigned char Port)
功能描述：控制电源控制器
参    数：FunNum -- 功能指示，当为 3 时发复位命令，为 1 继电器接通，为 0 继电器断开
          NetID -- 网络 ID
          Port  -- 继电器号
*********************************************************************************/
void Output_Relay485(unsigned char FunNum,unsigned char NetID,unsigned char Port){
     if (FunNum == 3){//主机发复位指令，格式：FF 02 03 00
         Send00485();//发 00 且第九位为 0 以分开各命令
         Sendbyte485(0xff);
         Sendbyte485(0x02);
         Sendbyte485(0x03);
         Sendbyte485(0x00);
     }
     else{//主机发信息至电源控制器（控制从机开关状态）,格式：ID 01 Value
         Delay(10);
         Sendbyte485(NetID);
         Sendbyte485(0x01);
         Sendbyte485(FunNum ? Port:(Port | 0x80));
     }
}
/******************************************************************************
函数原形：void Send00485(void)
功能描述：发分隔符,主机向网络设备发 00 Hex 以分隔开各个命令
```

```
参    数：无
***************************************************************************/
void Send00485(void){
    rULCON1 = 0x3b;              //bit 9 is 0
    rGPHDAT |= (0x1 << 7);       //GPH7/485 receive disable,send enable
    Delay(1);                    //一定要加，否则会出现从机接收不稳定
    while (!(rUTRSTAT1 & 0x6));  //Wait until THR is empty.
    WrUTXH1(~0x00);
    while(!(rUTRSTAT1 & 0x6));   //Wait until THR is empty.
    rGPHDAT &= ~(0x1 << 7);      //GPH7/485 receive enable
    rULCON1 = 0x33;              //bit 9 is 1
}
/***************************************************************************
函数原形：void Sendbyte485(unsigned char data)
功能描述：向 485 发一个字节的数据
参    数：data -- 要发送的数据
***************************************************************************/
void Sendbyte485(unsigned char data){
    rGPHDAT |= (0x1 << 7);       //GPH7/485 receive disable,send enable
    Delay(1);                    //一定要加，否则会出现从机接收不稳定
    while (!(rUTRSTAT1 & 0x6));  //Wait until THR is empty.
    WrUTXH1(~data);
    while (!(rUTRSTAT1 & 0x6));  //Wait until THR is empty.
    Delay(5);                    //慢速从机接收需要一定时间
    rGPHDAT &= ~(0x1 << 7);      //GPH7/485 receive enable
}
#define   FIND_DEVICE_WAIT_TIME       0xffff    //每个 ID 等待时间
#define   FIND_DEVICE_INFO_LENGTH     40        //输出信息的最大长度
#define   FIND_DEVICE_DATA_LENGTH     16        //接收回应的最大长度
#define   SERIES   "CRPWR-4"
#define   NET_ID    0x06
/***************************************************************************
函数原形：char FindDevice(unsigned char id, char * series_buf)
功能描述：查找网络设备
参    数：id -- 要查找的设备 ID
          buf-- 指向找到的设备续号
返    回：非 0 时为设备续号的长度,包括字符串最后的 NULL
***************************************************************************/
char FindDevice(unsigned char id, char * series_buf){
    unsigned int waitdata = 0;
    unsigned char j,i = 0,len;
    char buf[FIND_DEVICE_DATA_LENGTH];
    Send00485();        //分隔符
    Sendbyte485(0xff);  //查找网络设备命令：FF 02 03 01 ID 00
```

```
    Sendbyte485(0x02);
    Sendbyte485(0x03);
    Sendbyte485(0x01);
    Sendbyte485(id);
    Sendbyte485(0x00);
    while (++waitdata < FIND_DEVICE_WAIT_TIME){
    if (rUTRSTAT1 & 0x1){
        buf[i] = ~RdURXH1();
        if (i < FIND_DEVICE_DATA_LENGTH){
            i++;
        }
        else{
            break;
        }
        waitdata = 0;
    }
 }
 if (i > 8){
     for (j = 0; j < i; j++){
         if ((buf[j] == 0x02) && (buf[j + 2] == 0x05) && (buf[j + 3] == 0x0) && buf[j + 4]
== 0x0) {//从机返回主机的指令
             if ((len = buf[j+1] - 2) == sizeof(SERIES)){
                 if (strncmp(&buf[j+5], SERIES, buf[j+1] - 3) == 0){
                     for (i = 0; i < len - 1; i++){
                         *series_buf++ = buf[j+5+i];
                     }
                     *series_buf = 0;
                     return len;
                 }
             }
         }
     }
    }
    return 0;//fail
}
/****************************************************************************
函数原形：void main(void)
功能描述：485 测试程序
参    数：无
****************************************************************************/
void main(void){
     static unsigned int temp;
     static unsigned char i=1, j;
     char buf[FIND_DEVICE_DATA_LENGTH];
```

```
        ChangeClockDivider(1,1);          // 1:2:4
        ChangeMPllValue(0xa1,0x3,0x1);    // FCLK=202.8MHz
        rGPHCON = rGPHCON & ~(0x3 << 14) | (0x1 << 14);//setup 485 receive enable pin
        rGPHCON = rGPHCON & ~(0xf << 8) | (0xa << 8);  //setup TXD1 RXD1 pin
        Delay(0);
        Uart1_Init(0, 38400);
        FindDevice(NET_ID, buf);
        Output_Relay485(3, NET_ID, 0);
        while (1){
            Delay(50);
            Send00485();
            Sendbyte485(NET_ID);
            Sendbyte485(00);
            temp += 1;
            if ((temp % 100) == 0){
                Output_Relay485(i, NET_ID, j);
                j++;
                if (j > 3){
                    j = 0;
                }
                if (temp % 400 == 0){
                    if (i > 0) {
                        i = 0;
                    }
                    else{
                        i = 1;
                    }
                }
            }
        }
    }
```

1．2 出现的问题

主机控制电源控制器（从机）继电器吸合和断电时，有时控制不住。即 485 通信不稳定，从机接收有时会出错。用示波器两个探头分别测主机的发送（MAX485 的 pin4 DI）和从机的接收（MAX485 的 pin1 RO）发现：从机在接收到主机数据的第 1 个起始位以前一直都处于低电平，空闲的高电平时间太短。

原因是当主机使能接收、禁止发送（/RE_S = DE_S = 0）时，从机 RO_S 被拉低，总线禁止了从机接收。当主机禁止接收、重新使能发送（/RE_S = DE_S = 1）时，从机 RO_S 被拉高，总线使能从机接收。此时主机不能马上发送，应延时等待从机的 RO_S 恢复稳定到空闲状态。该时间应该大于 4 个位时间，如波特率为 38400 时，该延时时间应大于(1/38400) * 4 > 100us。当发送完数据后需要延时 300us（最好 500us）之后才能禁止发送（/RE=0）；否则从机不能正确接收数据！

2. Linux 驱动程序

```
……
#define IO_MAJOR    236
#define BUFFSIZE    16
#define nREDE       GPIO_H7      //485 接口芯片的接收、发送使能脚
#define TXD         GPIO_H4
#define RXD         GPIO_H5
static char rec_buf[BUFFSIZE], index, Rx_counter;//中断接收缓冲区，索引值及总接收总数
static void RS485_Init(void){     //用于 485 的串口寄存器及相关 GPIO 口设置
    set_gpio_ctrl(nREDE | GPIO_PULLUP_EN | GPIO_MODE_OUT);//setup 485 receive enable pin
    set_gpio_ctrl(TXD | GPIO_PULLUP_EN | GPIO_MODE_UART);//TXD1
    set_gpio_ctrl(RXD | GPIO_PULLUP_DIS | GPIO_MODE_UART);//RXD1
    UFCON1=0x27;                  //Rx Trigger 12byte,Tx Rx FIFO reset enable
    UMCON1=0x0;                   //AFC disable
    ULCON1=0x33;                  //8byte，stop 1，parity force 1
    UCON1=0x385;                  //rx=level,tx=level,enable timeout int,disable rx error
int.,normal,tx rx interrupt or polling
    UBRDIV1=80;                   //38400
}
static void Send00485(void){      //485 网络，发送第 9 位为 0 的分隔 0x00，作为命令串的开始
    ULCON1 = 0x3b;                //bit 9 is 0
    write_gpio_bit(nREDE, 1);     //GPH7/485 receive disable,send enable
    udelay(100);                  //必须加这延时，以等待从机
    while (!(UTRSTAT1 & 0x6));    //Wait until THR is empty.
    UTXH1 = ~0x00;                //485 协议中，都以反码发送
    while(!(UTRSTAT1 & 0x6));     //Wait until THR is empty.
    ULCON1 = 0x33;                //bit 9 is 1
}

static void Sendbyte485(unsigned char data, unsigned char end){//485 网络，发送一个字节
    while (!(UTRSTAT1 & 0x6));    //Wait until THR is empty.
    UTXH1 = ~data;
    if (end){                     //laster byte
        while (!(UTRSTAT1 & 0x6));//Wait until THR is empty.
        udelay(500);              //发送完最后一字节后，也需等待从机
        write_gpio_bit(nREDE, 0);//485 receive enable
    }
}
char Rec485(unsigned char counter, char *user_buf){
    unsigned long timeout = 2 + counter/30;//接收超时等待
    unsigned int jiffies1;
    index = 0;
    Rx_counter = counter;
```

```
    enable_irq(IRQ_RXD1);        //启用中断接收
    jiffies1 = jiffies;
    while ((jiffies < jiffies1+timeout) && (index < Rx_counter));//等待接收完成或超时退出
    disable_irq(IRQ_RXD1);
    if (index > 0){
        if (index < counter){
            copy_to_user(user_buf, rec_buf, index);
            return index;
        }
        else{
            copy_to_user(user_buf, rec_buf, counter);
            return counter;
        }
    }
    return 0;
}
static void rx_irq_handle(int irq, void *dev_id, struct pt_regs *regs){//中断接收数据
    char ch;
    while ((UFSTAT1 & 0xf) > 0){                        //Rx FIFO Count
        if (UERSTAT1){                                  //err
            ch = ~URXH1;
        }
        else {
            if (index < BUFFSIZE) {//在有时刚运行程序时就出现 OOP 错误时所加的判断
                rec_buf[index++] = ~URXH1;
            }
            if ((index == 2) && (rec_buf[0] == 0x02)){  //从机发出的命令字节数
                Rx_counter = rec_buf[1] + 2;
            }
        }
    }
}
static ssize_t RS485_read(struct file *filp, char *buf, size_t count, loff_t *f_pos){
    if (count > BUFFSIZE){
        printk("<1> buf size invalidation\n");
        return -EINVAL;
    }
    return (Rec485(count, buf));
}
static ssize_t RS485_write(struct file *filp, const char *buf, size_t count, loff_t *f_pos){
    unsigned char WriteBuf[BUFFSIZE];
    unsigned char i;
    if (count > BUFFSIZE) {
        printk("<1> buf size invalidation\n");
```

```
        return -EINVAL;
    }
    copy_from_user(WriteBuf, buf, count);
    Send00485();
    for (i = 0; i < (count - 1); i++){
        Sendbyte485(WriteBuf[i], 0);
    }
    Sendbyte485(WriteBuf[i], 1);
    return count;
}
//cmd[7:0]：命令，cmd[15:8]：从机 ID，cmd[23:16]：缓冲区尺寸，arg：指向数据缓冲区的指针
static int RS485_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned
long arg){
    unsigned char temp;
    unsigned char Len, i;
    char ioctl_buf[BUFFSIZE];
    int ret = 0;
    Len = (cmd >> 16) & 0xff;
    if (Len == 0) {
        return -EINVAL;
    }
    switch (cmd & 0xff){
        case 0:                                //reset:FF 02 03 00
                udelay(700);
                Send00485();
                Sendbyte485(0xff, 0);
                Sendbyte485(0x02, 0);
                Sendbyte485(0x03, 0);
                Sendbyte485(0x00, 1);
                break;
        case 1:                                //find device:FF 02 03 01 ID 00
                Send00485();
                Sendbyte485(0xff, 0);
                Sendbyte485(0x02, 0);
                Sendbyte485(0x03, 0);
                Sendbyte485(0x01, 0);
                Sendbyte485((cmd >> 8) & 0xff, 0);
                Sendbyte485(0x00, 1);
                ret = Rec485(Len, (char *)arg);
                break;
        case 2:                                //read 485 net command string, ID 00
                Send00485();
                Sendbyte485((cmd >> 8) & 0xff, 0);//ID
                Sendbyte485(0x00, 1);           //00
```

```
                ret = Rec485(Len, (char *)arg);
                break;
        case 3:                                //read string，可替代系统读函数
                ret = Rec485(Len, (char *)arg);
                break;
        case 4:                                //write string，可替代系统写函数
                Send00485();
                copy_from_user(ioctl_buf, (char *)arg, Len);
                for (i = 0; i < (Len - 1); i++){
                    Sendbyte485(ioctl_buf[i], 0);
                }
                Sendbyte485(ioctl_buf[i], 1);
                break;
        default:
                return -EINVAL;
    }
    return ret;
}
static struct file_operations RS485_fops={
 owner:  THIS_MODULE,
 write:  RS485_write,
 read:   RS485_read,
 ioctl:  RS485_ioctl,
};
static devfs_handle_t devfs_handle;
static int __init RS485_init(void){
    unsigned long flags;
    devfs_handle = devfs_register(NULL,DEVICE_NAME,DEVFS_FL_DEFAULT,IO_MAJOR,0,S_IFCHR |
S_IRUSR | S_IWUSR,&RS485_fops,NULL);
    RS485_Init();
    request_irq(IRQ_RXD1, rx_irq_handle, 0, DEVICE_NAME"_IRQ_UART1", NULL);
    disable_irq(IRQ_RXD1);
    printk(DEVICE_NAME ":Init OK\n");
    return 0;
}
static void __exit RS485_exit(void){
    devfs_unregister(devfs_handle);
}
module_init(RS485_init);
module_exit(RS485_exit);
```

3．驱动测试程序

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <fcntl.h>
unsigned char command[6] = {
    0xFF, 0x02, 0x03, 0x01, 0x06, 0x00     //find devices
};
int main(int argc,char *argv[]){
    unsigned char port;
    unsigned int i;
    char buf[16], outbuf[16];
    char ret;
    int fd;
    pid_t pid, pid1;
    fd = open("/dev/PGM_RS485",O_RDWR);
    if (fd < 0){
        perror("open device PGM_RS485 faill!");
        exit(1);
    }
//    write(fd, command, 6);          //该写查找设备命令
//    ret = read(fd, buf, 16);        //读设备信息，该两句等同于下面的 ioctl 查找设备命令
    ret = ioctl(fd, 0x100601, buf);   //查找 ID = 06 的 485 网络设备
    if ((ret > 5) && (ret < 16)){
        if ((buf[0] == 0x02) && (buf[2] == 0x05) && (buf[4] == 0x00)){
            buf[buf[1] + 2] = '\0';
            printf("find device:%s\n", &buf[5]);//打印设备信息
            ioctl(fd, 0x010600, buf);           //复位 485 设备命令
        }
        else{
            printf("no find device!\n");
        }
    }
    if ((pid = fork()) < 0){
        printf("fork err\n");
    }
    else if (pid == 0){               //father
        buf[0] = 0x06;                //ID
        buf[1] = 0x01;                //01
        buf[2] = 0x00;
        while (1){
            buf[2] ^= 0x80;
            write(fd, buf, 3);
            sleep(1);
        }
```

```
    }
    else{                              //children，定时轮寻网络设备，读设备信息
        while (1){
            for (i = 0; i < 10000; i++);
            ret = ioctl(fd, 0x100602, buf);//ID 00
        }
        close(fd);
    }
    close(fd);
    return 0;
}
```

4. 出现的问题

1> **read、write 系统函数中计算参数 buf 的长度——sizeof(buf)的结果不等于 buf 实际长度而一直为 4**

原因：当一个数组名作为参数传递给函数时，它就转换成了一个指针，而指针的长度为 4。如果不经过参数传递，则 sizeof(buf)的结果就为 buf 实际的长度

2> **485 网络控制需注意**

主机发送时要注意：当使能发送（/RE=1）时，要延时 100us 之后才能开始发送数据；当发送完数据后需要延时 300us 之后才能禁止发送（/RE=0）；否则从机不能正确接收数据（这于具体的从机有关)!

要使能接收的 FIFO，否则可能会出现个别数据丢失的情况当波特率为 38400 时 UBRDIV1=81 时现象很明显，UBRDIV1=80 时少些但还有，这可能是操作系统调度等较占用时间，来不及处理中断引起，当设置成 FIFO（UFCON1=0x27）后，即使 UBRDIV1=81 也都没有出现过丢码现象。

3> **要知道内核（驱动属于内核空间）没有多进程**，只有一个进程，这在用户空间是不同的。在用户空间里，可以使一个进程执行 while(1)，其它的进程也能跑，但是内核中不行，除非自己放弃（如调用 sleep_on 等一系列挂起及阻塞的函数）。

4> **Linux 2.4.x为不可抢占的内核，在单处理器（非SMP）环境下，自旋锁什么都不做，自旋锁类型 spinlock_t设置为空：**

```
typedef struct{
} spinlock_t;
```

相应地，自旋锁的操作函数不进行任何实质性处理。

```
#define spin_lock_init(lock) do { } while(0)
#define spin_lock(lock) (void)(lock) /* Not "unused variable". */
#define spin_is_locked(lock) (0)
#define spin_trylock(lock) ({1; })
#define spin_unlock_wait(lock) do { } while(0)
#define spin_unlock(lock) do { } while(0)
```

5> **在接收中断处理程序中，执行 wake_up_interruptible(&rx_queue)，为何没有立即唤醒等待队列中的进程。**程序及执行结果及打印信息：

进程中代码：

```
Enable_irq(IRQ_RXD1);
Printk(“<1> rx_queue sleep_on\n”);
Interruptible_sleep_on_timeout(&rx_queue, WAITTIME);
Printk(“<1> rx_queue sleep_up\n”);//还需要测试，在这之前禁止中断会是何结果？
```

```
Disable_irq(IRQ_RXD1);
IRQ_RXD1 中断代码：
If(index >= Rx_counter){
    Printk("<1> index is %d, Rx_counter is %d\n", index, Rx_counter);//多次打印该信息
    Wake_up_interruptible(&rx_queue);//接收完指定的数据后唤醒等代中的进程
}
#./RS485_test
RS485: Init OK
rx_queue sleep_on
index is 12, Rx_counter is 12
index is 13, Rx_counter is 12
index is 14, Rx_counter is 12
rx_queue sleep_up
```

因为等待对列都是在内核调度时处理的，所以当执行 Wake_up_interruptible(&rx_queue)后只有等到下次调度时才能唤醒，而在这之前仍有可能会产生中断。

6> **为何执行 local_irq_save(flags)不能禁用中断？代码如下，它仍能产生中断。**

```
enable_irq(IRQ_RXD1);
local_irq_save(flags);    //禁止中断
interruptible_sleep_on_tiimeout(&rx_queue, WAITTIME);
local_irq_restore(flags);
disable_irq(IRQ_RXD1);
```

原因是 interruptible_sleep_on_timeout()函数，它的一系列原型如下：

```
long interruptible_sleep_on_timeout(wait_queue_head_t *q, long timeout){
    SLEEP_ON_VAR
    current->state = TASK_INTERRUPTIBLE;
    SLEEP_ON_HEAD
    timeout = schedule_timeout(timeout);   //超时后返回
    SLEEP_ON_TAIL                          //下述定义
    return timeout;
}
#define   SLEEP_ON_TAIL                         \
wq_write_lock_irq(&q->lock);               \
__remove_wait_queue(q, &wait);                  \
wq_write_unlock_irqrestore(&q->lock,flags);
#define wq_write_unlock_irqrestore spin_unlock_irqrestore
#define spin_unlock_irqrestore(a,b)        {restore_flags((b));}
```

从上述可以看出当超时唤醒时，执行了一次 restore_flags。

另外当禁止中断后 jiffies 等都不会有变化，因为系统定时器的中断也被禁止了。

7> **ARM 中的 local_irq_save(flags)，执行该宏定义后，2410 的中断屏蔽寄存器（INTMSK，SUBINTMSK）值是不会变的，**它是通过控制 ARM 程序状态寄存器——CPSR 中的 I 和 F 来禁止 IRQ 和 FIQ 中断的。Flags 保存的是当的 CPSR 值。另外当执行该宏定义后，由于所有中断（包括系统定时器中断）都被禁止了，所以系统不会再产生调度，jiffies 值也被停止递增。

8> **485 总线冲突**，即当某一进程对 485 总线操作（接收从机数据）还没有完成，另一进程也对总线进行操作（发送数据）。如在 RS485_write()函数中发送分隔符语句（Send00485();）后面增加一条打印信

息（printk(“RS485_write Send00485 ”);)，在 Rec485()函数的等待接收数据进行睡眠的前后也增加打印信息，如下：

```
Enable_irq(IRQ_RXD1);
Printk(“<1> rx_queue sleep_on  ”);
Interruptible_sleep_on_timeout(&rx_queue, WAITTIME);
Printk(“<1> rx_queue sleep_up\n”);//还需要测试，在这之前禁止中断会是何结果？
Disable_irq(IRQ_RXD1);
```

按照事先的想法打印的信息应该一系列的是：

```
rx_queue sleep_on  rx_queue sleep_up
rx_queue sleep_on  rx_queue sleep_up
……
```

但在运行的过程中偶然会发现下述打印信息：

```
rx_queue sleep_on  RS485_write Send00485 rx_queue sleep_up
```

说明在接收还没有完成时，接收的进程进入了睡眠，此时内核重新调度执行了发送的进程，因此引起了 485 总线的竞态。

9> 485 总线竞态的防止

曾尝试过自旋锁等一系列的处理互斥问题的方法，但最终都不理想。最后由于考虑到内核空间的单进程特性，也就是说如果驱动程序自己不放弃（不调用 sleep_on 等的一系列挂起函数）内核，内核是不会调度去执行其它进程的，因此也就根本不会有其它进程对总线操作的可能。所以将 Interruptible_sleep_on_timeout(&rx_queue, WAITTIME)修改成忙操作如下：

```
enable_irq(IRQ_RXD1);                                    //启用中断接收
jiffies1 = jiffies;
while ((jiffies < jiffies1+timeout) && (index < Rx_counter));//等待接收完成或超时退出
disable_irq(IRQ_RXD1);
```

虽然这样比原先占用了一些时间资源，但绝对安全，所以值得。

八．红外学习与发射

1．硬件测试程序

```
#include "..\Inc\2410addr.h"
#include "..\Inc\2410lib.h"
#include "..\Inc\option.h"
#define  IR_Addr    (*(volatile unsigned char *)0x080000c8)
#define  IR_NUM_MAX  68
//红外脉冲的最大输入个数,正常:2(Lead code)+32(16bit custom*2)+32(16bit data*2)+1(end)=67
#define  IR_NUM_MIN  30
unsigned char gSending;      //红外发射标志，1 为发射
unsigned char gLearning;
//红外学习标志,0:未学习状态,1:表示进入红外学习,正等待红外信号输入 2:红外学习过程中
unsigned short gCount;
//红外学习过程中,对红外数据进行计数(定时器溢出时递增,输入脉冲跳沿中断是也递增)
unsigned short gLearnTime;   //红外学习时间,即输入脉冲高你电平的总个数
unsigned char gState;
```

```
    unsigned short gIRData[100]; //红外学习时,数据缓冲区,用于存放红外脉冲的高低电平时间,第 1
个数据为低电平时间
    unsigned short *pIRData;     //红外发送时,指向数据缓冲区
    unsigned char gTimeoutCount; //用于红外学习时的定时器溢出计数器
    unsigned char gState_Study;
    unsigned char gPWMEn;        //38KHz 载波输出使能，1 为开定时器 1 输入输出载波信号，0 为停止
    unsigned short gLen;         //要发送红外的数据长度
    void LearnOK(void);
    void __irq Int_IRStudy(void);
    void __irq Int_Timer0(void);
    void __irq Int_Timer1(void);
    /*********************************************************************************
    函数原形：void Init_IR(void)
    功    能：初始化红外相关的 I/O,中断及定时器等
    *********************************************************************************/
    void Init_IR(void){
        rGPFCON = rGPFCON & ~(0x3 << 0) | (0x2 << 0);  //EINT0=GPF0
        rGPFUP = rGPFUP | ~(0x1 << 0);                 //disable
        rGPBCON = rGPBCON & ~(0x3 << 2) | (0x2 << 2);  //TOUT1
        rEXTINT0 = rEXTINT0 & ~(0x7 << 0) | (0x6 << 0);//EINT0:Both edge triggered
        pISR_EINT0 = (int)Int_IRStudy;
        ClearPending(BIT_EINT0);
        pISR_TIMER0 = (int)Int_Timer0;
        rTCFG0 = rTCFG0 & ~(0xff << 0) | (0x01 << 0);  //prescale value = 1
        rTCFG1 = rTCFG1 & ~(0xf << 0) | (0x2 << 0);    //divider value = 8
        rTCNTB0 = 0xffff;
        rTCFG1 = rTCFG1 & ~(0xf << 4) | (0x2 << 4);//divider value = 8,50MHz/(1+1)/8=3.125MHz
        rTCNTB1 = 0x52;                                 //3.125MHz/38K=82=0x52;
        rTCMPB1 = 0x29;                                 //0x52/2=0x29
        rTCON = rTCON & ~(0xf<<8)|(0xa<<8);//Timer1:auto reload; Update TCNTB1,TCMPB1; Stop
        pISR_TIMER1 = (int)Int_Timer1;
        rINTMSK &= ~(BIT_EINT0);                        //打开外部中断 0 开始红外学习
        gState = 1;
        gLearning = 0;
    }
    /*********************************************************************************
    函数原形：void __irq Eint3Isr(void)
    功    能：IR 输入脉冲双边沿中断，记录电平时间宽度
    *********************************************************************************/
    void __irq Int_IRStudy(void){
        rGPFCON = rGPFCON & ~(0x3 << 0) | (0x0 << 0);   //GPF0=IN
        if (gLearning == 0){
            if (!(rGPFDAT & 0x1)){                      //确认是否为低电平
                gLearning = 1;
```

```
            rTCON = rTCON & ~(0x1f << 0) | 0x2;     //Updata TCNTB0
            rTCON = rTCON & ~(0x1f << 0) | 0x1;     //start for timer0
            rINTMSK &= ~(BIT_TIMER0);               //打开定时器中断
            gState = 0;                             //保存当前脉冲电平值
            pIRData = gIRData;                      //初始化红外数据缓冲区
        }
    }
    else if (gLearning == 1){                       //确认红外引导码
        *pIRData = rTCNTO0;
        if ((rGPFDAT & 0x1) && (gState == 0) && (*pIRData < 45535) && (*pIRData > 25535)){
                                        //Lead Code 9ms, > 6.4ms, < 12.8ms
            rTCON = rTCON & ~(0x1f << 0) | 0x2;     //Updata TCNTB0
            rTCON = rTCON & ~(0x1f << 0) | 0x1;     //start for timer0
            gLearning = 2;
            gState = 1;
            pIRData++;
        }
        else{                                       //Lead Code check fail
            rINTMSK |= BIT_TIMER0;                  //关定时器中断
            rTCON = rTCON & ~(0x1f << 0) | 0x2;     //stop for timer0
            gLearning = 0;
        }
    }
    else{
        if (((rGPFDAT & 0x1) && gState == 0) || ((!(rGPFDAT & 0x1)) && gState == 1)){
            *pIRData++ = rTCNTO0;//记录计数器值,第1个数据为低电平时间,第2为高电平时间。。。
            gState = gState ? 0 : 1;                //乒乓开关用于状态翻转判定
            if (gLearning > IR_NUM_MAX){
                LearnOK();
            }
            else{
                rTCON = rTCON & ~(0x1f << 0) | 0x2;
                rTCON = rTCON & ~(0x1f << 0) | 0x1;
                gLearning++;
            }
     }
    }
    rGPFCON = rGPFCON & ~(0x3 << 0) | (0x2 << 0);   //GPF0=EINT0
    ClearPending(BIT_EINT0);
}
/*****************************************************************************************
函数原形：void __irq Timer0Done(void)
功    能：红外学习定时器溢出中断处理,根据红外学习脉冲数来确定学习是否成功结束
*****************************************************************************************/
```

```
void __irq Int_Timer0(void){
    if (gSending == 1){
        if(gCount >= gLen){
            gSending = 0;                              //发送结束
            gPWMEn = 0;
          rTCON = rTCON & 0xffff000;                   //stop for timer0 timer1
          rINTMSK |= BIT_TIMER1 | BIT_TIMER0;          //关定时器中断
        }
        else{
            if (gPWMEn == 0) {                    //上一个数据为低电平,则该数据为高电平
                gPWMEn = 1;
                rTCNTB0 = 0xffff - *pIRData;           //高电平时间
                rTCON = (rTCON&0xffffff0) | 0x2;
                rTCON = (rTCON&0xffffff0) | 0x1;
                rTCON = (rTCON&0xffff0ff) | 0xa00;
                rTCON = (rTCON&0xffff0ff) | 0x900;  //重新启动 38K 载波输出
            }
            else{
                gPWMEn = 0;
                rTCNTB0 = 0xffff - *pIRData;           //低电平时间
                rTCON = (rTCON & 0xffffff0) | 0x2;
                rTCON = (rTCON & 0xffffff0) | 0x1;
            }
            pIRData++;
            gCount++;
        }
    }
    else if (gLearning > IR_NUM_MIN) {                 //红外学习时间溢出结束
         LearnOK();
    }
    else{
        rINTMSK |= BIT_TIMER0;                         //关定时器中断
        rTCON = rTCON & ~(0x1f << 0) | 0x0;            //stop for timer0
        gLearning = 0;
    }
    ClearPending(BIT_TIMER0);
}
/*********************************************************************************
函数原形：void __irq Timer1Done(void)
功    能：产生红外的 38KHz 载波
参    数：gPWMEn -- 1 为输出 38K 载波,0 为停止输出
*********************************************************************************/
void __irq Int_Timer1(void){
    if (!gPWMEn){
```

```
        rTCON = rTCON & 0xffff0ff;                      //stop for Timer 1
    }
    ClearPending(BIT_TIMER1);
}

/*********************************************************************************
函数原形：void LearnOK(void)
功    能：红外学习成功
*********************************************************************************/
void LearnOK(void){
    rTCON = rTCON & ~(0x1f << 0) | 0x2;         //stop timer0
    rINTMSK |= BIT_EINT0 | BIT_TIMER0;
    gLen = gLearning - 1;                       //保存红外学习的数据个数
    gLearning = 0xff;
}
/*********************************************************************************
函数原形：void SendIR(void)
功    能：启动红外发送
*********************************************************************************/
void SendIR(void){
    pIRData = gIRData;
    gSending = 1;
    gCount = 0;
    gPWMEn = 1;                                 //发送高电平，与红外学习时电平相反
    rTCNTB0 = 0xffff - *pIRData;
    pIRData++;
    gCount++;
    rTCON = (rTCON & 0xffffff0) | 0x2;          //Update TCNTB0, TCMPB0
    rTCON = (rTCON & 0xffffff0) | 0x1;          //Start for Timer 0
    rTCON = (rTCON & 0xffff0ff) | 0xa00;//Timer1:auto reload;Update TCNTB1,TCMPB1; Stop
    rTCON = (rTCON & 0xffff0ff) | 0x900;        //Start for Timer 1
    rINTMSK &= ~(BIT_TIMER0 | BIT_TIMER1);
}
/*********************************************************************************
函数原形：void LearnIR(unsigned char which)
功    能：置 IR 学习开始条件,即进入红外学习
*********************************************************************************/
void LearnIR(void){
    unsigned char i = 0;
    IR_Addr = 0x0f;                             //使能所有 IR 通道输出
    while (1){
        Init_IR();
        while (gLearning != 0xff);              //等待红外学习完成
        SendIR();
```

```
        while (gSending);                      //等待红外发射完成
    }
}
```

2．Linux 驱动程序

```
#define DEVICE_NAME    "PGM_IR"
#define BUZZ_MAJOR      236
#define IRQ_IRInput     IRQ_EINT0
#define IR_NUM_MAX      68   //2(Lead code)+32(16bit custom*2)+32(16bit data*2)+1(end)=67
#define IR_NUM_MIN      30
#define Wait_Time       1800 //IR study wait times 18s
#define pIR_ADDR        0x080000c8
void * vIR_ADDR;
unsigned char Learning, IO_State;                            //IR study use
unsigned char PWMEn, Transmit_flag, TLen, Transmit_Index;    //IR Transmits
static DECLARE_WAIT_QUEUE_HEAD(Wait_Study);
static DECLARE_WAIT_QUEUE_HEAD(Wait_Transmit);
static void pwm_irq_handle(int irq, void *dev_id, struct pt_regs *regs) {//38K
    if(!PWMEn){
        TCON = TCON & (~(0xf << 8)) | (0xa << 8);
                   //Timer1:auto reload;Inverter off;Update TCNTB1,TCMPB1;stop
    }
}
static void timer0_Transmit_irq_handle(int irq, void *dev_id, struct pt_regs *regs){
    unsigned short *pIRData = dev_id;
    if (Transmit_flag == 1) {                       //IR Transmit
        if(Transmit_Index >= TLen){
            Transmit_flag = 0;
            PWMEn = 0;
            TCON = TCON & 0xffff000;                //stop for timer0 timer1
            wake_up_interruptible(&Wait_Transmit);
        }
        else{
            if (PWMEn == 0){
                PWMEn = 1;
                TCNTB0 = 0xffff - *(pIRData + Transmit_Index);
                TCON = (TCON & 0xffffff0) | 0x2;
                TCON = (TCON & 0xffffff0) | 0x1;
                TCON = (TCON & 0xffff0ff) | 0xa00;
                TCON = (TCON & 0xffff0ff) | 0x900;
            }
            else{
```

```
                PWMEn = 0;
                TCNTB0 = 0xffff - *(pIRData + Transmit_Index);
                TCON = (TCON & 0xfffff0) | 0x2;
                TCON = (TCON & 0xfffff0) | 0x1;
            }
            Transmit_Index++;
        }
    }
}
static void timer0_Study_irq_handle(int irq, void *dev_id, struct pt_regs *regs){
    TCON = TCON & ~(0x1f << 0);                 //stop for timer0
    if (Learning > IR_NUM_MIN){
        wake_up_interruptible(&Wait_Study);
    }
    else{
        Learning = 0;
    }
}
static void IRInput_irq_handle(int irq, void *dev_id, struct pt_regs *regs){
    unsigned short *pIRData = dev_id;
    set_gpio_ctrl(GPIO_F0 | GPIO_PULLUP_DIS | GPIO_MODE_IN); //INPUT
    if (Learning == 0){
        if (!read_gpio_bit(GPIO_F0)) {         //check Lead code
            Learning = 1;
            TCON = TCON & ~(0x1f << 0) | 0x2;  //Updata TCNTB0
            TCON = TCON & ~(0x1f << 0) | 0x1;  //start for timer0
            IO_State = 0;
        }
    }
    else if (Learning == 1){                   //affirm Lead code
        *(pIRData) = TCNTO0;
        if (read_gpio_bit(GPIO_F0) && (IO_State == 0) && (*pIRData < 45535) && (*pIRData >
25535)) {                                      //Lead Code 9ms, > 6.4ms, < 12.8ms
            TCON = TCON & ~(0x1f << 0) | 0x2;  //Updata TCNTB0
            TCON = TCON & ~(0x1f << 0) | 0x1;  //start for timer0
            Learning = 2;
            IO_State = 1;
        }
        else {                                 //Lead Code check fail
            TCON = TCON & ~(0x1f << 0);        //stop for timer0
            Learning = 0;
        }
    }
    else{
```

```
            if((read_gpio_bit(GPIO_F0) && IO_State == 0) || ((!read_gpio_bit(GPIO_F0)) &&
IO_State == 1)) {
                *(pIRData+Learning-1) = TCNTO0;
                IO_State=IO_State?0:1;
                if (Learning > IR_NUM_MAX){
                    wake_up_interruptible(&Wait_Study);
                }
                else{
                    TCON = TCON & ~(0x1f << 0) | 0x2;
                    TCON = TCON & ~(0x1f << 0) | 0x1;
                    Learning++;
                }
            }
        }
        set_gpio_ctrl(GPIO_F0 | GPIO_PULLUP_DIS | GPIO_MODE_EINT); //EINT0
    }
    static ssize_t IR_Study(struct file *filp, char *buf, size_t count, loff_t *f_pos){
        int ret, i;
        unsigned short * Study_Buf;
        if ((count < IR_NUM_MIN*2) || (count > 256)){
            printk("<1> buf size invalidation\n");
            return -EINVAL;
        }
        if (!(Study_Buf = kmalloc(count, GFP_KERNEL))){
            printk("<1> kmalloc fail\n");
            return -EINVAL;
        }
        ret = request_irq(IRQ_IRInput, IRInput_irq_handle, SA_INTERRUPT,
DEVICE_NAME"IRQ_EINT0", Study_Buf);
        if (ret != 0){
            printk("<1> IR busy\n");
            kfree(Study_Buf);
            return -EBUSY;
        }
        ret = request_irq(IRQ_TIMER0, timer0_Study_irq_handle, SA_INTERRUPT,
DEVICE_NAME"IRQ_TIMER0_Study", NULL);
        if (ret != 0){
            printk("<1> IR busy\n");
            kfree(Study_Buf);
            free_irq(IRQ_IRInput, Study_Buf);
        }
        Learning = 0;
        set_gpio_ctrl(GPIO_F0 | GPIO_PULLUP_DIS | GPIO_MODE_EINT);
        interruptible_sleep_on_timeout(&Wait_Study, Wait_Time);
```

```
        free_irq(IRQ_IRInput, Study_Buf);
        free_irq(IRQ_TIMER0, NULL);
        if (Learning > IR_NUM_MIN){
    /*        for (i = 0; i < Learning - 1; i++)
            {
                if (!(i%10)) printk("\n<1>");
                printk("%04x,", *(Study_Buf+i));
            }
            printk("\nLearning is %d\n", Learning);*/
            copy_to_user(buf, Study_Buf, (ret = (Learning - 1) * 2));
        }
        else{
            ret = 0;
        }
        kfree(Study_Buf);
    //    free_irq(IRQ_IRInput, Study_Buf);
    //    free_irq(IRQ_TIMER0, NULL);
        return ret;
    }
    static ssize_t IR_Transmit(struct file *filp, const char *buf, size_t count, loff_t *f_pos){
        int ret;
        unsigned short * Transmit_Buf;
        if ((count < IR_NUM_MIN*2) || (count > 256)){
            printk("<1> buf size invalidation\n");
            return -EINVAL;
        }
        if (!(Transmit_Buf = kmalloc(count, GFP_KERNEL))){
            printk("<1> kmalloc fail\n");
            return -EINVAL;
        }
        copy_from_user(Transmit_Buf, buf, count);
        ret    =    request_irq(IRQ_TIMER0,    timer0_Transmit_irq_handle,    SA_INTERRUPT,
DEVICE_NAME"IRQ_TIMER0", Transmit_Buf);
        if (ret != 0){
          printk("<1> IR busy\n");
            kfree(Transmit_Buf);
            return -EBUSY;
        }
        ret = request_irq(IRQ_TIMER1, pwm_irq_handle, SA_INTERRUPT, DEVICE_NAME"IRQ_TIMER1",
NULL);
        if (ret != 0){
            printk("<1> IR busy\n");
            kfree(Transmit_Buf);
            free_irq(IRQ_TIMER0, Transmit_Buf);
```

```
        return -EBUSY;
    }
    Transmit_flag = 1;
    PWMEn = 1;
    TLen = count/2;
    TCNTB0 = 0xffff - *Transmit_Buf;
    Transmit_Index = 1;
    TCON = (TCON & 0xfffffff0) | 0x2;         //Update TCNTB0, TCMPB0
    TCON = (TCON & 0xfffffff0) | 0x1;         //Start for Timer 0
    TCON = (TCON & 0xffff0ff) | 0xa00;
     //Timer1:auto reload; Inverter off; Update TCNTB1,TCMPB1; Stop
    TCON = (TCON & 0xffff0ff) | 0x900;       //Start for Timer 1
    interruptible_sleep_on_timeout(&Wait_Transmit,20);//9+4.5+2.24*32+0.56=85.74
    TCON &= 0xffff000;
    free_irq(IRQ_TIMER0, Transmit_Buf);
    free_irq(IRQ_TIMER1, NULL);
    kfree(Transmit_Buf);
    return count;
}
static int IR_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long
arg){
    static unsigned char channel = 0;
    if (arg < 4){
        switch (cmd){
            case 0:
                    channel &= ~(0x1 << arg);
                    break;
            case 1:
                    channel |= (0x1 << arg);
                    break;
            case 2:
                    channel = 0;
                    break;
            case 3:
                    channel = 0xf;
                    break;
            default:
                    break;
        }
        writeb(channel, vIR_ADDR);
    }
}
static struct file_operations IR_fops = {
    owner:THIS_MODULE,
```

```
    write:IR_Transmit,
    read: IR_Study,
    ioctl:IR_ioctl,
};
static devfs_handle_t devfs_handle;
static int __init IR_init(void){
    devfs_handle = devfs_register(NULL, DEVICE_NAME, DEVFS_FL_DEFAULT, BUZZ_MAJOR, 0,
S_IFCHR | S_IRUSR | S_IWUSR, &IR_fops, NULL);
    set_gpio_ctrl(GPIO_B1 | GPIO_PULLUP_DIS | GPIO_MODE_TOUT);//TOUT1
    set_external_irq(IRQ_IRInput, EXT_BOTH_EDGES, GPIO_PULLUP_DIS);
    TCFG0 |= TCFG0_PRE0(1);                    //Timer0,1 prescale value = 1
    TCFG1 = TCFG1 & (~(0xff << 0)) | (0x2 << 0) | (0x2 << 4);
     //Timer0,1 divider value = 8, 50MHz/(1+1)/8=3.125MHz
    TCNTB0 = 0xffff;                           //21ms
    TCNTB1 = 0x52;                             //3.125MHz/38K=82=0x52;
    TCMPB1 = 0x29;                             //0x52/2=0x29
    TCON &=  ~(0xfff << 0);                    //Timer0,1 Stop
    Learning = 0;
    Transmit_flag = 0;
    PWMEn = 0;
    vIR_ADDR = ioremap(pIR_ADDR, 1);
    printk(KERN_INFO DEVICE_NAME ": init OK\n");
    return(0);
}
static void __exit IR_cleanup(void){
    iounmap(vIR_ADDR);
    devfs_unregister(devfs_handle);
}
module_init(IR_init);
module_exit(IR_cleanup);
MODULE_LICENSE("GPL");
```

3．驱动测试程序

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <fcntl.h>
int main(int argc,char *argv[]){
    int fd, IR_fd;
    int ret, i;
    unsigned short buf[100], TBuf[100];
    fd = open("/dev/PGM_IR", O_RDWR);
```

```
    if (fd < 0){
        perror("open PGM_IR fail");
        exit(1);
    }
    ioctl(fd, 3, 0);
    if (argc == 2){
        IR_fd = open("/tmp/IR_fd0", O_RDONLY);
        if (IR_fd < 0){
            perror("open IR_fd0");
        }
        else{
            ret = read(IR_fd, TBuf, sizeof(TBuf));
            if (ret > 0){
                for (i = 0; i < ret/2; i++){
                    if (!(i%10)) printf("\n");
                    printf("%04x,", TBuf[i]);
                }
            }
            write(fd, TBuf, ret);
        }
    }
    ret = read(fd, buf, sizeof(buf)) / 2;
    if (ret > 0){
        for (i = 0; i < ret; i++){
            if (!(i%10)){
                printf("\n");
            }
            printf("%04x,", buf[i]);
        }
        IR_fd = creat("/tmp/IR_fd0", O_RDWR);
        if (IR_fd > 0){
            printf("creat IR_fd sucess\n");
            write(IR_fd, buf, sizeof(short) * ret);
        }
        else{
            perror("IR_fd");
        }
    }
    else{
        perror("read PGM_IR");
    }
    close(fd);
    return 0;
}
```

测试方法：

1> 测试红外学习时直接运行可执行文件即可，然后按住红外遥控，学习完后，将打印出红外学习数据，然后创建”IR_fd0”文件，将数据保存。

2> 测试红外发射，在执行文件时，需给传送任一参数，如“# ./IR_test 1”。它将 IR_fd0 中的红外数据发射出去。

4．出现的问题

用“request_irq(IRQ_TIMER0, timer0_Transmit_irq_handle, SA_INTERRUPT, DEVICE_NAME"IRQ_TIMER0", Transmit_Buf);”申请中断时，但用“free_irq(IRQ_TIMER0, NULL);”不能释放中断(可用“#cat /proc/interrupts”命令观察)。

必须使 free_irq 中的 dev_id 参数与 request_irq 中的该参数保持一至，即：free_irq(IRQ_TIMER0, Transmit_Buf);

5．总结

此驱动与上面几个驱动有几点不同，主要表现在对中断的处理

1> **申请中断和释放中断都直接在 read(红外学习)、write(红外发射)系统函数里完成**。原因主要考虑到红外学习与发射都需要阻塞等待(sleep_on 的系列函数)，但在阻塞的过程中又不允许其它进程也对红外进行操作（因为它们都占用了相同的硬件资源，如定时器 0 等)。所以为了实现红外驱动的互斥，当进行应用用户调用红外学习或发射时才申请中断，如果中断申请失败，说明已被占用，则出错退出。

2> **同一中断有两个不同的中断处理程序**。红外学习时，调用 request_irq(IRQ_TIMER0, timer0_Study_irq_handle, SA_INTERRUPT, DEVICE_NAME"IRQ_TIMER0_Study", NULL);红外发射时调用 request_irq(IRQ_TIMER0, timer0_Transmit_irq_handle, SA_INTERRUPT, DEVICE_NAME"IRQ_TIMER0", Transmit_Buf);这样比共用一个处理程序的可读性和效率更高。

3> **在申请中断时 request_irq 中的 void *dev_id 参数作为私有数据**（红外数据指针）传递给中断处理程序的该参数，参考上面各中断处理程序。

九．网络编程

1．常用函数

下面总结记录了《UNIX 网络编程》一书中，我觉得比较常用的一些函数、概念，更详细的请参考该书。

1．1 套接口地址结构：sockaddr_xx

IPv4 套接口地址结构通常也称为“网际套接口地址结构” ：sockaddr_in

```
struct in_addr{
in_addr_t  s_addr;              //32-bit IPv4 address
                                //network byte ordered
};
struct sockaddr_in{
```

```
uint8_t             sin_len;
sa_family_t         sin_family;   //AF_INET
in_port_t           sin_port;     //16-bit TCP or UDP port number
                                  //network byte ordered
struct  in_addr_t   sin_addr;     //32-bit Ipv4 address
                                  //network byte ordered
char           sin_zero[8];       //unused
}
```

通用套接口地址结构：

```
struct  sockaddr{
uint8_t        sa_len;
sa_family_t    sa_family ;        //address family :AF_xxx value
char           sa_data[14];       //protocol specific address
}
```

由于套接口函数被定义为采用指向通用套接口地址结构的指针，如bind函数原型所示：

```
int  bind(int, struct sockaddr *, socklen_t);
```

所以套接口函数的任何调用都必须将指向特定于协议的套接口地址结构的指针类型转换成指向通用套接口地址结构的指针。例如：

```
struct  sockaddr_in  serv;        //IPv4 socket address structure
bind(sockfd, (struct sockaddr *)&serv, sizeof(serv));
```

另外，在unp.h中有如下定义：

```
#defined      SA       struct sockaddr
```

所以可以简略的写成：

```
Bind(sockfd, (SA *)&serv, sizeof(serv));
```

1．2 值-结果参数

当把套接口地址结构传递给套接口函数时，总是通过指针来传递的。结构的长度也作为参数来传递，其传递方式取决于结构的传递方向：从进程到内核，还是从内核到进程。

从进程到内核传递套接口地址结构有3个函数：bind、connect和sendto。它们结构的长度以整数大小传递：

```
struct sockaddr_in serv;
connect(sockfd, (SA *)&serv, sizeof(serv));
```

从内核到进程传递套接口地址结构有4个函数：accept、recvfrom、getsockname和getpeername。它们结构的长度以整数大小的指针传递：

```
struct sockaddr_un cli;           //Unix domain
socklen_t len ;
len=sizeof(cli) ;                 //len is a value
getpeername(unixfd, (SA *)&cli, &len);
```

为何将结构大小由整数改为指向整数的指针呢？这是因为：当函数被调用时，结构大小是一个值(value，此值告诉内核该结构的大小，使内核在写此结构时不至于越界)，当函数返回时，结构大小又是一个结果(result，它告诉进程内核在此结构中确切存储了多少信息)，这种参数类型叫做**值-结果**(value-result)参数。

1．3 字节排序函数

内存中存储多个字节有“小端”和“大端”两种字节序。我们把系统所用的字节序称为主机字节序(host byte order)，而网络协议中的称为网络字节序(network byte order)，它们不一定相同，所以就有了主机字节序和网络字节序间的相互转换函数：

```
＃include <netinet/in.h>
uint16_t htons(uint16_t host16bitvalue);
uint16_t htonl(uint32_t host32bitvalue);
```

均返回：网络字节序值

```
uint16_t ntohs(uint16_t net16bitvalue);
uint16_t ntohl(uint32_t net32bitvalue);
```

均返回：主机字节序值

1．4 字节操纵函数

多字节字段的操作有两组函数。第一组函数以字母 b(表示 byte)打头，起源于 4.2BSD，由支持套接口函数的系统提供，如下：

```
#include <strings.h>
void bzero(void *dest, size_t nbytes);
void bcopy(const void *src, void *dest, size_t nbytes);
void bcmp(const void *ptr1, const void *ptr2, size_t nbytes);
```

返回：0 —— 相等，非 0 —— 不相等

第二组函数其名字以 men(表示 memory)打头，起源于 ANSI C 标准，由支持 ANSI C 库的系统提供，如下：

```
#include <string.h>
void *memset(void *dest, int c, size_t len);
```

与 bzero 不同的是：将目标指定数目的字节置为值 c

```
void *memcpy(void *dest, const void *src, size_t nbytes);
```

与 bcopy 相比交换了两个指针参数的顺序，memcpy 两个指针的顺序是按照 C 的赋值语句从左到右的顺序书写的 ：

```
dest = src ;
int memcmp(const void *ptr1, const void *ptr2, size_t nbytes);
```

返回：0 —— 相同，>0 或<0 —— 不相同，具体>0 还是<0 取决于第一个不等字节的大小。

另外 memXXX 函数都要求有一个长度参数，且它总是最后一个。

1．5 地址转换函数 —— ASCII 字符串与网络字节序的二进制间转换地址

inet_aton、inet_addr 和 inet_ntoa 在点分十进制数串(例如，“202.168.112.96”)与它的 32 位网络字节序二进制值间转换 IPv4 地址，如下：

```
#include <arpa/inet.h>
int inet_aton(const char *strptr, struct in_addr *addrptr);
//返回：1——串有效，0——串无效
```

in_addr_t inet_addr(const char *strptr);

//返回：若成功，返回 32 位二进制的网络字节序地址；若出错，则返回 INADDR_NONE(一般为一个 32 位均为 1 的值)，这意味着点分十进制数串 255.255.255.255（IPv4 的有限广播地址）不能由此函数处理，因为它的二进制值被用来指示函数的失败。**目前一般由 inet_aton 替代它。**

char *inet_ntoa(struct in_addr inaddr);

//返回：指向点分十进制数串的指针

下面两个函数较新，对 IPv4 和 IPv6 地址都能处理。

#include <arpa/inet.h>

int inet_pton(int family, const char *strptr, void *addrptr);

//返回：1 成功，0 输入不是有效的表达格式，-1 出错

const char *inet_ntop(int family, const void *addrptr, char *strptr, size_t len);

//返回：指向结果的指针——成功，NULL——出错

参数 len 是目标的大小，以免函数溢出其调用者的缓冲区，如果太小，无法容纳表达格式结果（包括终止的空字符），则返回一个空指针，并置 errno 为 ENOSPC。

另外在头文件<netinet/in.h>中有如下定义：

#define INET_ADDRSTRLEN 16

#define INET6_ADDRSTRLEN 46

1．6 字节流套接口函数

字节流套接口(如 TCP 套接口)上的 read 和 write 函数所表示的行为不同于通常的文件 I/O。字节流套接口上的读或写输入或输出的字节数可能比要求的数量少，但这不是错误状况，原因是内核中套接口的缓冲区可能已达到了极限。此时需要再次调用 read 或 write 函数，以输入或输出剩余的字节。

ssize_t readn(int filedes, void *buff, size_t nbytes);

ssize_t written(int filedes, const void *);

ssize_t readline(int filedes, void *buff, size_t maxien);

原码参考 P74 页

```
ssize_t writen(int fd, const void *vptr, size_t n){
 size_t        nleft;
 ssize_t       nwritten;
 const char    *ptr;
 ptr = vptr;
 nleft = n;
 while (nleft > 0) {
      if ( (nwritten = write(fd, ptr, nleft)) <= 0){
           if (nwritten < 0 && errno == EINTR)
                nwritten = 0;      /* and call write() again */
           else
                return(-1);             /* error */
      }
      nleft -= nwritten;
      ptr   += nwritten;
 }
```

```
 return(n);
}
```

recv 和 send 函数类似于标准的 read 和 write 函数，不过需要一个额外的参数，如下：

```
#include <sys/socket.h>
ssize_t recv(int sockfd, void *buff, size_t nbytes, int flags);
ssize_t send(int sockfd, const void *buff, size_t nbytes, int flags);
```

//返回：读入或写出字节数——成功，-1——出错

前 3 个参数等同于 read 和 write 的 3 个参数，flags 参数的值或为 0，或为下述一个或多个常值的逻辑或。

MSG_DONTROUTE (send) —— 绕过路由表查找

MSG_DONTWAIT (recv 和 send) —— 仅本操作非阻塞

MSG_OOB (recv 和 send) —— 发送或接收带外数据

MSG_PEEK (recv) —— 窥看外来消息

MSG_WAITALL (recv) —— 等待所有数据，它告知内核不要在尚未读入请求数目的字节之前让一个读操作返回。出现下述情况例外：1> 捕获一个信号 2> 连接被终止 3> 套接口发生一个错误。

1．7 基本套接口函数

所有客户和服务器都从调用 socket 开始，它返回一个套接口描述字。客户随后调用 connect 函数来建立与服务器的连接，服务器则调用 bind、listen 和 accept。套接口通常使用标准的 close 函数关闭，不过有时也会用 shutdown 关闭。另外还有获得本地协议地址(getsockname)和远地协议地址(getsockname)等函数。

无论是客户还是服务器，为了执行网络 I/O，必须做的第一件事情就是调用 sock 函数，指定期望的通信协议类型来创建套接口描述字。

```
#include <sys/socket.h>
int socket(int family, int type, int protocol);
```

//返回：非负描述字(套接口描述字，简称套接字：sockfd)——成功，-1——出错

family 为下述常值，指明协议族(family)：

AF_INET —— IPv4 协议

AF_INET6 —— IPv6 协议

AF_LOCAL —— Unix 域协议

AF_ROUTE —— 路由套接口

AF_KEY —— 密钥套接口

type 为下述常值，指明套接口类型：

SOCK_STREAM —— 字节流套接口

SOCK_DGRAM —— 数据报套接口

SOCK_SEQPACKET —— 有序分组套接口

SOCK_RAW —— 原始套接口

Protocol 为下述常值，指明某个协议类型，或设为 0，以选择 family 和 type 组合的系统缺省值。

IPPROTO_TCP —— TCP 传输协议

IPPROTO_UDP —— UDP 传输协议

IPPROTO_SCTP —— SCTP 传输协议

客户端调用connect函数来建立与服务器的连接，如下：

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
//返回：0 —— 成功，-1 —— 出错
```

sockfd是由socket函数返回的套接口描述字；servaddr为必须含有所连接服务器IP和端口号的套接口地址结构指针；addrlen为servaddr结构的大小。

如果是TCP客户在调用connect函数将激发TCP的三路握手过程，而且仅在连接建立成功或出错是才返回。另外，如果connect失败则该套接口不再可用，必须close该套接口描述字，重新调用socket。

bind函数把一个本地协议地址赋予一个套接口

```
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
```

sockfd是由socket函数返回的套接口描述字；myaddr是一个指向特定于协议的地址结构指针；addrlen是该地址结构的长度。

本地地址结构参数：

sin_family：协议族

sin_port：服务器在启动时必须捆绑它们的众所周知端口，只有这样才能被大家(客户端)认识。

sin_addr.s_addr：对于TCP客户，这就为在该套接口上发送的IP数据报指派了源IP地址。对于TCP服务器，这就限定该套接口只接收那些目的地为这个IP地址的客户连接。

调用bind函数可以指定IP地址(本地地址)或端口，也可以不指定IP地址（通配地址）和端口。对于IPv4，通配地址由常值INADDR_ANY来指定，它允许服务器在任意网络接口上接受客户连接（假定服务器主机有多个网络接口）。

listen函数仅由TCP服务器调用，通常应该在调用socket和bind两个函数之后，并在调用accept函数之前调用，它做两件事情：

当socket函数创建一个套接口时，它被假设为一个主动套接口，也就是说，它是一个将调用connect发起连接的客户套接口，指示内核应接受指向该套接口的连接请求。调用listen导致套接口从CLOSED状态转换到LISTEN（被动）状态。

listen函数的第二个参数规定了内核应该为相应套接口排队的最大连接个数。

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
返回：0 —— 成功，-1 —— 出错
```

accept函数由TCP服务器调用，用于从已完成连接队列队头返回下一个已完成连接。如果已完成连接队列为空，那么进程被投入睡眠（假定套接口为缺省的阻塞方式）。

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
返回：非负描述字 —— 成功，-1 —— 出错
```

参数cliaddr和addrlen用来返回已连接的对端进程(客户端)的协议地址。addrlen是值-结果参数。如果accept成功，那么其返回值是由内核自动生成的一个全新描述字，代表与所返回客户的TCP连接。在讨论accept函数时，我们称它的第一个参数为监听套接口（listening socket）描述字(由socket创建，随后用作bind和listen的第一个参数的描述字)，称它的返回值为已连接套接口(connected socket)描述字。一个服务器通常仅仅创建一个监听套接口，它在该服务器的生命期内一直存在。内核为每个由服务器进程接受的客户连接创建了一个已连接套接口（也就是说对于它的TCP三路握手过程已经完成）。当服务器完成对于某个给定客户的服务时，相应的已连接套接口就被关闭。如果对返回客户协议地址不感兴趣，那么可以把cliaddr和addrlen均置为空指针。

得本地协议地址（getsockname）和远地协议地址（getsockname）函数

```
#include <sys/socket.h>
int getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen);
int getpeername(int sockfd, struct sockaddr *peeraddr, socklen_t *addrlen);
```

两者均返回：0 —— 成功，-1 —— 出错

wait 和 waitpid 等待子进程终止函数

```
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

两者均返回：终止的进程 ID 为成功，或为-1 出错

statloc 指向子进程的终止状态，如果不关心终止状态，则可指定为空指针。

pid 参数允许我们指定想等待的进程 ID，值-1 表示等待第一个终止的子进程。

option 允许指定附加选项，最常用的选项是 WNOHANG，它告知内核在没有已终止子进程时不要阻塞。

下述为等待多个子进程终止的例子：

```
#include "unp.h"
void sig_chld(int signo){
    pid_t pid;
    int stat;
    while ((pid = waitpid(-1, &stat, WNOHANG)) > 0){
        printf("child %d terminated\n", pid);
    }
    return;
}
```

shutdown 函数，它与 close 函数相比两点不同：1> close 把描述字的引用计数值减 1，仅在该计数变为 0 时才关闭套接口。使用 shutdown 可以不管引用计数就激发 TCP 的正常连接终止。2> close 终止数据传送的两个方向：读和写。而 shutdown 可以只关闭一半的 TCP 连接，即只终止读连接或写连接。

```
#include <sys/socket.h>
int shutdown(int sockfd, int howto);
```

返回：0 —— 成功，-1 —— 出错

函数的行为依赖于 howto 参数的值：

SHUT_RD —— 关闭连接的读这一半，套接口中不再有数据可接收，而且套接口接收缓冲中的现有数据都被丢弃。进程不能再对这样的套接口调用任何读函数。对一个 TCP 套接口这样调用 shutdown 函数后，由该套接口接收的来自对端的任何数据都被确认，然后悄然丢弃。

SHUT_WR —— 关闭连接的写这一半，当前留在套接口发送缓冲区中的数据将被发送掉，后跟 TCP 的正常连接终止序列。进程不能再对这样的套接口调用任务写函数。

SHUT_RDWR —— 连接的读这一半和写这一半都关闭。

1．8 I/O 复用：select 和 poll 函数

阻塞型 I/O 只能等待一个描述字就绪，即进程调用系统调用（如：accept，read 等）后阻塞直到数据报到达且被拷贝到应用进程的缓冲区中或者发生错误才返回。而有了 I/O 复用，就可以调用 select 或 poll，阻塞在这(select 和 poll)两个系统调用中的某一个之上，而不是阻塞在真正的 I/O 系统调用，这样就可以等待多个描述字就绪。

select 函数：

```
#include <sys/select.h>
```

```
#include <sys/time.h>
int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset, const struct timeval *timeout);
```

返回：就绪描述字的正数目，0 —— 超时，-1 —— 出错

timeout 参数告知内核等待所指定描述字中的任何一个就绪可花多长时间，其结构如下：

```
struct timeval{
 long tv_sec;  //seconds
 long tv_usec; //microseconds
}
```

timeout 取值有三种可能：1> 空指针，永远等待下去，直到有一个描述字准备就绪。2> 结构中的定时器值为 0，根本不等待，即检查描述字后立即返回。3> 结构中的定时器值大于 0，等待一段固定时间。

参数 readset、writeset 和 exceptset 指定要让内核测试读、写和异常条件的描述字，如果对某一个条件不感兴趣，可以将它设为空指针，表示不测试该类型（或读，或写，或异常）的任何描述字，如下述 select 函数只测试读就绪的描述字：select(maxfdp1, &rset, NULL, NULL, NULL);

给这三个参数的每一个指定一个或多个描述字值隐藏在名为 fd_set 的数据类型和以下四个宏中：

```
void FD_ZERO(fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_CLR(int fd, fd_set *fdset);
void FD_ISSET(int fd, fd_set *fdset);
```

maxfdp1 参数指定待测试的描述字个数，它的值是待测试的最在描述字加 1（因此把该参数命名为 maxfdp1），描述字 0、1、2......一直到 maxfdp1 – 1 均将被测试。

poll 函数：在功能上与 select 函数类似，不过在处理流设备时，它能够提供额外的信息。

```
#include <poll.h>
int poll(struct pollfd *fdarray, unsigned long nfds, int timeout);
```

返回：就绪描述字的正数目，0 —— 超时，-1 —— 出错

fdarray 参数指向 pollfd 结构数组第一个元素的指针，每个数组元素都是一个 pollfd 结构，用于指定测试某个给定描述字 fd 的条件。

```
struct pollfd{
 int fd;       //descriptor to check
 short events; //events of interest on fd
 short revents;     //events that occurred on fd
}
```

测试的条件由 events 成员指定，函数在相应的 revents 成员中返回该描述字的状态。下述为 events 和 revents 标志的常值：

POLLIN ——普通或优先级带数据可读，POLLRDNORM 和 POLLRDBAND 逻辑或
POLLRDNORM ——普通数据可读
POLLRDBAND ——优先级带数据可读
POLLPRI ——高优先级数据可读
POLLOUT ——普通数据可写
POLLWRNORM ——普通数据可写，等同于 POLLOUT
POLLWRBAND ——优先级带数据可写
POLLERR ——发生错误
POLLHUP ——发生挂起
POLLNVAL ——描述字不是一个打开的文件

nfds 参数指定结构数组中元素的个数

timeout 参数指定 poll 函数返回前等待多长时间，取值如下：

INFTIM：永远等待；0：立即返回，不阻塞进程；>0：等待指定数目的毫秒数。

1．9 信号处理

信号(signal)就是通知某个进程发生了某个事件，有时也称软件中断(software interrupt)。信号通常是异步发生的，也就是说进程预先不知道信号准确发生的时刻。它可以由一个进程发给另一进程（或自身），也可以由内核发给某个进程。

SIGCHLD 信号就是由内核在任何一个进程终止时发给它的父进程的一个信号。

```
Sigfunc *signal(int signo, Sigfunc *func){
    struct sigaction    act, oact;
    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (signo == SIGALRM) {
#ifdef    SA_INTERRUPT
        act.sa_flags |= SA_INTERRUPT; /* SunOS 4.x */
#endif
    }
    else{
#ifdef    SA_RESTART
        act.sa_flags |= SA_RESTART;        /* SVR4, 44BSD */
#endif
    }
    if (sigaction(signo, &act, &oact) < 0){
        return(SIG_ERR);
    }
    return(oact.sa_handler);
}
/* end signal */
Sigfunc *Signal(int signo, Sigfunc *func) {     /* for our signal() function */
    Sigfunc    *sigfunc;
    if ( (sigfunc = signal(signo, func)) == SIG_ERR){
        err_sys("signal error");
    }
    return(sigfunc);
}
```

如在服务器中当子进程关闭时，捕获 SIGCHLD，对其处理以防止僵死进程。信号处理函数：

```
void sig_chld(int signo){
    pid_t pid;
    int stat;
    while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0){//当没有信号处理时返回-1 退出循环
        printf("child %d terminated\n", pid);
```

```
    }
    return;
}
```

要在创建第一个子进程之前调用：Signal(SIGCHLD, sig_chld);

1．10 调试命令

```
#netstat －a —— 检查服务器监听套接口的状态，命令显示如下：
#netstat －a |grep 9877
Active Internet connections (servers and established)
Proto Recv-Q   Send-Q   Local Addres  Foreign Address   State
tcp          0        0    *：9877       *：*            LISTEN
```

上例为只显示与端口号 9877 相关的状态，Local Addres 为本地地址结构，Foreign Addres 为远程地址结构。“*”表示一个为 0 的 IP 地址（INADDR_ANY，通配地址），或为 0 的端口号。State 为网络正处于的状态，LISTEN 为正在监听，另外还有 ESTABLISHED 表示已确定连接的，TIME_WAIT 等待超时等，可参考 TCP 状态转换图。

2．服务器程序

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <signal.h>
#include <errno.h>
#define SERV_PORT 9877
#define MAXLINE 256
struct SERIAL_DATA{
    unsigned short Length;
    unsigned short wReserved;
    unsigned char *pBuffer;
};
struct DOWN_LOAD_PORT{
    unsigned short CommandNum;
    unsigned short MaxLength;
    struct SERIAL_DATA *psCommand;
    struct SERIAL_DATA sOldInput;
    struct SERIAL_DATA sOldHttpInput;
    struct SERIAL_DATA sCommandHead;
    struct SERIAL_DATA sDelimiter;
    struct SERIAL_DATA sAckOK;
    struct SERIAL_DATA sAckDetect;
```

```
        struct SERIAL_DATA sAckERR;
        unsigned char byParity;
        unsigned char bReserved;
        unsigned short wReserved;
    };
    const char *gpCommandHead="\x02\x1b\x43";
    const char *gpSerialCommand[] = {
        "\x08","\xcf\xad\xf0\xe7\xd9\xb6\xe0\xf8",//0  update user program
        "\x08","\xfc\xad\xf0\xe7\xd9\xb6\xe0\x8f",//1  update system program
        "\x04","INFO",                           //2  look over user information
        "\x03","VER",                            //3  look over system version
        "\x07","Control",                        //4  switch run times show
        "\x07","IRLearn",                        //5  IR study
        "\x07","GetTime",                        //6  read time
        "\x07","SetTime",                        //7  set time
        "\x03","Del",                            //8  delete user program
        "\x0d","GSQRestartGSQ",                  //9  reboot user program
        "\x0f","GSQHardResetGSQ",                //10 systerm hardware reset
        "\x06","Search",                         //11 find devices for 485NET
        "\x01","\xf0",                           //12 terminal check command,test connecting
        "\x08","\xfc\xda\x67\x85\x2d\x7f\x9e\x88",//13 send IR code test
        "\x0c","SearchPlugin",                   //14 search plugin devices
        "\x05","Digit",                          //15 Digit Signal
        "\x06","Analog",                         //16 Analog Signal
        "\x06","Serial",                         //17 Serial Signal
        "\x06","IRLong",                         //18 IR study of long
        "\x0b","SetBaudrate",                    //19 setup baudrate
        "\x0c","ReadBaudrate",                   //20 read baudrate
    };
    const char * gpDelimiter="\x02\x0d\x0a";
    const char * gpAckOK= "\x08\xe7\xa8\xb6\xceOK\x0d\x0a";
    const char * gpAckERR="\x01\x9f";
    const char * gpAckDetect="\x01\x0f";
    static struct DOWN_LOAD_PORT *gpDownPort;
    void* InitCommand(void){
        int i;
        struct SERIAL_DATA *pSerial;
        gpDownPort = (struct DOWN_LOAD_PORT*)malloc(sizeof(struct DOWN_LOAD_PORT));
        gpDownPort->CommandNum = sizeof(gpSerialCommand)/8;
        gpDownPort->byParity = 0;
        pSerial = gpDownPort->psCommand = (struct SERIAL_DATA*)malloc(sizeof(struct SERIAL_DATA)
*gpDownPort->CommandNum);
        gpDownPort->MaxLength = 0;
        for (i = 0; i < gpDownPort->CommandNum * 2; pSerial++){
```

```
            pSerial->Length = (unsigned short)(**(gpSerialCommand+(i++)));
            pSerial->pBuffer = (unsigned char *)(*(gpSerialCommand+(i++)));
            if (gpDownPort->MaxLength < pSerial->Length){
                gpDownPort->MaxLength = pSerial->Length;
            }
        }
        gpDownPort->sCommandHead.Length = (unsigned short)(*gpCommandHead);
        gpDownPort->sCommandHead.pBuffer = (unsigned char *)gpCommandHead + 1;
        gpDownPort->sDelimiter.Length = (unsigned short)(*gpDelimiter);
        gpDownPort->sDelimiter.pBuffer = (unsigned char *)gpDelimiter + 1;
        gpDownPort->sAckOK.Length = (unsigned short)(*gpAckOK);
        gpDownPort->sAckOK.pBuffer = (unsigned char *)gpAckOK + 1;
        gpDownPort->sAckDetect.Length = (unsigned short)(*gpAckDetect);
        gpDownPort->sAckDetect.pBuffer = (unsigned char *)gpAckDetect + 1;
        gpDownPort->sAckERR.Length = (unsigned short)(*gpAckERR);
        gpDownPort->sAckERR.pBuffer = (unsigned char *)gpAckERR + 1;
        gpDownPort->MaxLength          +=          gpDownPort->sCommandHead.Length          +
gpDownPort->sDelimiter.Length;
        gpDownPort->sOldInput.Length = 0;
        gpDownPort->sOldInput.pBuffer = (unsigned char *)malloc(gpDownPort->MaxLength + 4);
        gpDownPort->sOldHttpInput.Length = 0;
        gpDownPort->sOldHttpInput.pBuffer = (unsigned char *)malloc(256);
        memset(gpDownPort->sOldHttpInput.pBuffer,0,256);
        return gpDownPort;
    }
    int Process_Command(int nIDCommand,int sockfd){          //命令处理函数
        struct SERIAL_DATA *pSerial = gpDownPort->psCommand + nIDCommand;
        int k;
        if ( (k = writen(sockfd, gpDownPort->sAckOK.pBuffer, gpDownPort->sAckOK.Length)) < 0){
            printf("write faill\n");
            return (-1);
        }
        printf("command %d:%s\n", nIDCommand, pSerial->pBuffer); //打印查找到的命令串
    }

    int Communication(int sockfd){                           //与客户机通信处理程序
        struct SERIAL_DATA *pSerial;
        ssize_t nread, nbuf, len, i;
        char buf[MAXLINE];
        char *ptr;
    again:
        ptr = buf;
        nbuf = 0;
        while (1){
```

```
            if ( (nread = recv(sockfd, ptr, MAXLINE, 0)) <= 0){//等待接收客户机数据
                if (errno == EINTR){
                    continue;
                }
                else{
                    perror("Communication: read error");
                    return (-1);
                }
            }
            else{
                nbuf += nread;
                ptr += nread;
                if (nbuf <= gpDownPort->sCommandHead.Length){//匹配命令头
                    if (memcmp(buf, gpDownPort->sCommandHead.pBuffer, nbuf) == 0){
                        continue;
                    }
                    else{                                   //命令头不匹配，应答出错
                        if(writen(sockfd,gpDownPort->sAckERR.pBuffer,
gpDownPort->sAckERR.Length) < 0){
                            return (-1);
                        }
                        else{
                            goto again;                     //清缓冲区重新开始接收
                        }
                    }
                }
                else     if     (nbuf     <=     (gpDownPort->sCommandHead.Length     +
gpDownPort->sDelimiter.Length)){
                    continue;
                }
                else {
                    if                    (memcmp(&buf[nbuf-gpDownPort->sDelimiter.Length],
gpDownPort->sDelimiter.pBuffer, gpDownPort->sDelimiter.Length) == 0){//匹配结束符
                        len     =     nbuf     -     gpDownPort->sCommandHead.Length     -
gpDownPort->sDelimiter.Length;
                        pSerial = gpDownPort->psCommand;
                        for (i = 0; i < gpDownPort->CommandNum; i++, pSerial++){//查找命令
                            if (len == pSerial->Length){
                                if            (memcmp(&buf[gpDownPort->sCommandHead.Length],
pSerial->pBuffer, pSerial->Length) == 0){
                                    Process_Command(i, sockfd);//命令处理
                                    goto again;
                                }
                            }
```

```
                    }
                    printf("Bad Command\n");
                    if          (writen(sockfd,          gpDownPort->sAckERR.pBuffer,
gpDownPort->sAckERR.Length) < 0){
                        return (-1);
                    }
                    else{
                        goto again;                      //无效命令，重新开始接收
                    }
                }
            }
        }
    }
}
int main(int argc, char *argv[]){
    struct sockaddr_in cliaddr, servaddr;               //客户和服务器地址结构
    int listenfd, connfd;                               //监听和连接描述字
    socklen_t clilen;                                   //客户地址结构长度
    pid_t childpid;                                     //子进程描述符
    if ( (listenfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0){ //创建套接口
        perror("call to socket");
        exit(1);
    }
    bzero(&servaddr, sizeof(servaddr));                 //服务器地址结构清 0
    servaddr.sin_family = AF_INET;                      //网络协议：IPv4 协议
    servaddr.sin_addr.s_addr = INADDR_ANY;              //通配地址
    servaddr.sin_port = htons(SERV_PORT);               //网络端口
    if (bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0){
        perror("call to bind");
        exit(1);
    }
    if (listen(listenfd, 20) < 0){
        perror("call to listen");
        exit(1);
    }
    printf("Accepting connections ...\n");
    Signal(SIGCHLD, sig_chld);                          //创建子进程结束信号处理
    InitCommand();                                      //初始化与客户机通信协议及命令
    while (1){
        clilen = sizeof(cliaddr);
        if ( (connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &clilen)) < 0){
            if (errno == EINTR){
                continue;
            }
```

```
            else{
                perror("call to accept");
                exit(1);
            }
        }
        if ( (childpid = fork()) == 0){                  // child process
            close(listenfd);                             // close listening socket
            Communication(connfd);                       //处理与客户机间的通信
            exit(0);
        }
        close(connfd);
    }
}
```

另外与信号相关及 writen 等函数均为上面一些小节中列出的程序。

3．测试用客户程序

可以在另一台 PC 机（或者本机的另一终端，指向 IP：127.0.0.1）的 linux 下运行下述客户程序进行测试，上面的服务器程序可以先在 PC 的 linux 下运行测试，也可以编译成在 ARM 板上运行，效果一样。

```
#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netdb.h>
#include <errno.h>
#define MAXLINE 256
#define SERV_PORT 9877
#define max(a,b) ((a) > (b) ? (a) : (b))
const char *gpCommandHead="\x02\x1b\x43";
const char *gpDelimiter="\x02\x0d\x0a";
const char *gpAckOK="\x08\xe7\xa8\xb6\xceOK\x0d\x0a";
const char *gpAckERR="\x01\x9f";
void str_cli(FILE *fp, int sockfd){
    int maxfdp1, stdineof;
    fd_set rset;
    char sendline[MAXLINE], recvline[MAXLINE];
    int n, i;
    stdineof = 0;
    FD_ZERO(&rset);
    while (1){
        if (stdineof == 0){
            FD_SET(fileno(fp), &rset);
        }
        FD_SET(sockfd, &rset);
        maxfdp1 = max(fileno(fp), sockfd) + 1;
```

```
        select(maxfdp1, &rset, NULL, NULL, NULL);
        if (FD_ISSET(sockfd, &rset)){
            if ( (n = read(sockfd, recvline, MAXLINE)) <= 0){
                return;
            }
            if (memcmp(recvline, gpAckOK+1, n) == 0){
                write(fileno(fp), "AckOK\n", 6);
            }
            else if (memcmp(recvline, gpAckERR+1, n) == 0){
                write(fileno(fp), "AckERR\n", 7);
            }
        }
        if (FD_ISSET(fileno(fp), &rset)){
            if ( (n = read(fileno(fp), sendline, MAXLINE)) == 0) {
                stdineof = 1;
                shutdown(sockfd, SHUT_WR);
                FD_CLR(fileno(fp), &rset);
                continue;
            }
            writen(sockfd, gpCommandHead+1, 2);
            writen(sockfd, sendline, n-1);
            writen(sockfd, gpDelimiter+1, 2);
        }
    }
}
int main(int argc, char *argv[]){
    int sockfd;
    struct sockaddr_in servaddr;
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(SERV_PORT);
    inet_pton(AF_INET, "192.168.1.1", &servaddr.sin_addr);
    connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr));
    str_cli(stdin, sockfd);
    exit(0);
}
```

4．利用 IO 复用替代多进程的并发服务器

如果连接服务器的客户端不是几个，而是几十个，此时再一一创建这么多进程，可想而知这需要多少资源。如果当一个客户改变了服务器某个设备的状态，而在其它客户上的该设备状态也要更新，这怎么办？利用进程间通信吗？此时我选择了利用 IO 复用来实现。

```
int SoftwareTPServer(int port){
```

```
struct sockaddr_in cliaddr, servaddr;
int listenfd, connfd, maxfd, nready, i, clientfd[MAX_CLIENT], maxi, msgqlen;
socklen_t clilen;
fd_set rset, allset;
msg_SoftwareTP pmsg;
InitCommand();
maxi = -1;
for (i = 0; i < MAX_CLIENT; i++){
    clientfd[i] = -1;
}
FD_ZERO(&allset);
if ( (listenfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0){
    perror("call to socket");
    exit(1);
}
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(port);
if (bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0){
    perror("call to bind");
    exit(1);
}
if (listen(listenfd, 20) < 0){
    perror("call to listen");
    exit(1);
}
printf("SoftwareTP Accepting connections ...\n");
FD_SET(listenfd, &allset);
FD_SET(g_Variant.qid_SoftwareTP, &allset);
if (listenfd > g_Variant.qid_SoftwareTP){
    maxfd = listenfd;
}
else{
    maxfd = g_Variant.qid_SoftwareTP;
}
while (1){
    rset = allset;
    nready = select(maxfd + 1, &rset, NULL, NULL, NULL);
    if (FD_ISSET(listenfd, &rset)){
        clilen = sizeof(cliaddr);
        if ( (connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &clilen)) < 0)
        {
            if (errno == EINTR){
```

```
                goto drop;
            }
            else{
                perror("call to accept");
                exit(1);
            }
        }
        for (i = 0; i < MAX_CLIENT && clientfd[i] > 0; i++);
        if (i < MAX_CLIENT){
            clientfd[i] = connfd;
            FD_SET(connfd, &allset);
            if (connfd > maxfd){
                maxfd = connfd;
            }
            if (i > maxi){
                maxi = i;
            }
        }
        else{
            printf("too many clients");
            close(connfd);
        }
drop:   if (--nready <= 0){
            continue;
        }
    }
    if (FD_ISSET(g_Variant.qid_SoftwareTP, &rset)){
        if  (  (msgqlen  =  msgrcv(g_Variant.qid_SoftwareTP,  &pmsg,  256,  type_out,
IPC_NOWAIT)) > 0)
        {
            for (i = 0; i <= maxi; i++){
                if (clientfd[i] > 0){
                    printf("msgqlen:%d,SoftwareTP out:%d\n", msgqlen, type_out);
                    writen(clientfd[i], pmsg.buf, pmsg.Length);
                }
            }
        }
        if (--nready <= 0){
            continue;
        }
    }
    for (i = 0; i <= maxi; i++){
        if (clientfd[i] < 0){
            continue;
```

```
            }
            if (FD_ISSET(clientfd[i], &rset)){
                if (Communication_SoftwareTP(clientfd[i], 10006) != 0){
                    printf("client[%d] had quit out.\n", i);
                    close(clientfd[i]);
                    FD_CLR(clientfd[i], &allset);
                    clientfd[i] = -1;
                }
            }
            if (--nready <= 0){
                break;
            }
        }
        if (i == maxi){
            for (; i >= 0; i--){
                if (clientfd[i] > 0){
                    break;
                }
            }
            maxi = i;
        }
    }
}
```

5．用无线网络测试上述程序

如果目标板和PC机的无线网络双方都配置成点对点成功（能ping），则它们与有线的以太网没有任何区别，可以分别正常运行服务器与客户机程序。另外 PC 机也可通过无线网络通过 ztelnet(telnet 或用windows下自带的超级终端登陆)。

十．系统时间的实现

Linux时间是由两个地方同时维护的，一个是RTC寄存器维护的硬件时间（或称硬件时钟），另一个是Linux内核维护的系统时间（或称系统时钟）。PC机的linux系统，在启动时将系统时钟与硬件时钟同步，在正常退出时将硬件时钟与系统时钟同步。也可以利用hwclock命令实现硬件时钟和系统时钟间的同步。

Linux内核提供给应用程序关于时间、日期的系列API函数都是针对系统时钟，date命令也是针对系统时钟的。在mizi linux下，修改完系统时钟后，在掉电前不会将硬件时钟与系统时钟进行同步，而mizi linux自带的busybox没有hwclock命令，必须重新编译busybox使其增加该命令。

实现方案一.在应用程序中利用内核提供的API函数（如time、stime、gettimeofday、settimeofday、mktime、localtime 等）查看和设置系统时钟，再使硬件时钟与系统时钟同步。如：1.在编译 busybox 时选上hwclock；2.在/etc/init.d/rcS里增加一行hwclock －s，让系统在启动时同步RTC；3.在设置完时间后，运行hwclock －w保存到RTC。

实现方案二．直接写驱动读写RTC，应用程序不管linux的系统时间。

实现方案三．写一个硬件时钟和系统时钟同步的驱动。

Mizi linux 内核中提供了 RTC 驱动，位于/dev/misc/rtc，源代码为 kernel/drivers/char/s3c2410-rtc.c。

目前我的做法是：系统启动时，从 RTC 驱动中读取硬件时钟来同步系统时钟，平时修改系统时钟（时时钟设置等）时，立刻写 RTC 驱动来同步硬件时钟。

另外，s3c2410a 与三星公司以前的其它处理器（如 s3c44b0x）不同的是寄存器 BCDDAY，44b0x 用该寄存器表示日期（范围为 1 ~ 31），而 2410 用它表示星期（范围为 1 ~ 7），用 BCDDATE 表示日期。但在 mizi linux 中没有做相应的修改，所以出现了错误。为了不修改其它的源代码，可以只在 kernel/include/asm-arm/arch-s3c2410/S3C2410.h 中两个寄存器的地址定义互换。如将原来的：

```
#define BCDDATE     bRTC(0x7c)
#define BCDDAY      bRTC(0x80)
```

换成

```
#define BCDDATE     bRTC(0x80)
#define BCDDAY      bRTC(0x7c)
```

十一．关于进程的体会

进程能少创建一个就少创建，因为它会浪费很多资源，如内核调度需要时间；子进程会对父进程的内存进行拷贝而花费大量的内存；特别是进程之间不共享全局变量，之间需要通过进程间的系列通信机制才能交互等缺点。

1．进程间不共享变量

在程序中定义了全局变量，在 main 函数中创建了一个子进程，但在父进程中对该变量所作的修改不会影响到子进程，同时对子进程的修改也不会对父进程的该变量产生任何影响。在创建子进程时，子进程对父进程的内存区做了完全的拷贝，且相互独立的占用两个不同的内存区，所以子进程的该变量值只是刚创建时父进程的当前值，以后就相互独立互不影响。可以用下述代码进行测试：

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
unsigned char val, val1;
int main(){
    pid_t pid;
    val1 = 0x44;
    if ( (pid = fork()) == 0){
        while (1){
            val = 0x77;
            printf("child:val=%02x, val1=%02x\n", val, val1);
            sleep(1);
            val = 0x55;
            printf("child:val=%02x, val1=%02x\n", val, val1);
            sleep(1);
        }
    }
    else{
```

```
        while (1){
            sleep(1);
            val1 = 0x99;
            printf("father:val=%02x, val1=%02x\n", val, val1);
            sleep(1);
            val1 = 0xaa;
            printf("father:val=%02x, val1=%02x\n", val, val1);
        }
    }
}
```

运行结果如下：

```
child:val=77, val1=44
father:val=00, val1=99
child:val=55, val1=44
father:val=00, val1=aa
child:val=77, val1=44
father:val=00, val1=99
```

2．进程通信——信号的使用

2．1 信号定义系统调用

```
#include <signal.h>
void (*signal(int signo, void (*func)(int)))(int);
```

返回：成功则指向以前的信号处理函数，若出错则为 SIG_ERR

signo 参数为信号名，func 为函数指针（即产生该信号时将调用的信号处理函数）

上述函数原型太过复杂，可以用 typedef 定义使其简单化。

```
typedef void Sigfunc(int);
```

然后 signal 函数原型可写成：

```
Sigfunc *signal(int signo, Sigfunc *func);
```

2．2 使用例子

```
#define SIG_HARDRESET 34                  //定义信号名
typedef void Sigfunc(int);
Sigfunc *signal(int signo, Sigfunc *func){
    struct sigaction act, oact;
    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (signo == SIGALRM){
#ifdef SA_INTERRUPT
        act.sa_flags |= SA_INTERRUPT;
```

```
#endif
    }
    else{
#ifdef    SA_RESTART
        act.sa_flags |= SA_RESTART;
#endif
    }
    if (sigaction(signo, &act, &oact) < 0){
        return(SIG_ERR);
    }
    return(oact.sa_handler);
}
Sigfunc *Signal(int signo, Sigfunc *func){
    Sigfunc *sigfunc;
    if ( (sigfunc = signal(signo, func)) == SIG_ERR){
        perror("signal error");
    }
    return(sigfunc);
}
void sig_softreset(int signo){ //信号处理函数
    static pid_t pid_UserSystem = 0;
    if (signo == SIG_SOFTRESET){
        if (pid_UserSystem){
            kill(pid_UserSystem, SIGKILL);
            pid_UserSystem = 0;
            printf("kill UserSystem\n");
        }
        if ( (pid_UserSystem = fork()) == 0){
           ......
            kill(getppid(), SIG_SOFTRESET);//在子进程中发信号调用处理函数
        }
    }
    return;
}
int main(int argc, char *argv[]){
    int ret;
    pid_t pid_InputElement, pid_TimeTick, pid_IrQueue, pid_UserSystem;
    Signal(SIG_SOFTRESET, sig_softreset);
    kill(getpid(), SIG_SOFTRESET);//在父进程中发信号调用处理函数
    ......
}
```

3. 防止僵死进程

当某种原因使父进程终止时，子进程将会变成僵死进程继续工作，而浪费各种资源，这不是我们所期望的。

在上述例子中加入一个信号处理函数，和在main函数中增加信号定义如下：

```
void sig_chld(int signo){//信号处理，当一个子进程结束时被调用
    pid_t pid;
    int stat;
    while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0){
        printf("child %d terminated\n", pid);
    }
    return;
}
Signal(SIGCHLD, sig_chld);//加入到上述的main()函数中，且在创建子进程前调用
```

第四阶段 用户图形界面设计

前一个项目没有图形界面，所以不需要GUI编程，但是我们另一系列的产品是有图形界面的，也够复杂，先前是用s3c2410裸奔（国外对手用Windows CE的较多，也有Windows XP Embedded和Linux） 。后来我们也用CE开发了一款，软件由VC程序员负责，硬件基本是照搬开发板商提供的原理图，要扩展其它模块时就增加一片MCU通过串口和主芯片及CE系统通信，很是被动，当然这不能怪CE，只能怪我们自己没有在CE下开发底层设备的能力。既然已经开始了Linux，那就先学习使用Linux下的GUI吧（等有时间了再突破wince的底层驱动等）！Linux下的GUI虽然有很多，但我只考虑QT，原因是用的人多，文档资料全。由于QT需要C++(嗨，在学校时没有学，应该有朋友和我一样的吧？)，我先是买了Bjarne Stroustrup的《C++程序设计语言》，看了前面部分，后来听说钱能的《C++程序设计教程》不错，也买了一本学习。有了C++基础后，就买了本倪继利的《QT及Linux操作系统窗口设计》，错啦！翻完了，还不知道如何开始一个最简单的程序（后来参考还是可以了解一些信息的），还是看QT文档吧！后来买了《C++ GUI QT3编程》，研读和编译运行了书上大部分的例程，QT安装目录下有丰富的例子都有源代码，非常的好。前期工作做好后，又要回到目标平台！建立开发环境，编译QTE、QTOPIA及应用程序，分别编译运行在宿主机和目标板等，理解整个开发过程。

一. QT应用编程

《C++ GUI QT3编程》和书上的例子已经足够了，再加上QT安装后自带的例程足以完成非常复杂编程的参考。

二. Qt/Embedded和QTOPIA

1. Linuette平台

1.1 How to use Virtual FrameBuffer

虚拟帧缓冲允许Qt/Embedded程序在宿主机Linux环境下运行， 即将Qt/Embedded应用程序编译为

x86 平台版本，使其可以在 X Window 下的 qvfb 窗口中运行。在应用程序调试好后，再编译为 ARM 平台版本，下载到嵌入式设备上运行。

1．1．1 Executing qvfb

#cd /linuette/host/Qt/X11/2.3.1/

#. ./env.sh　　　　　　　　　　　　　　　　　　//设置环境变量，‘.’与‘.’之间有空格

env.sh 文件内容如下：

export QTDIR=/linuette/host/Qt/X11/2.3.1

export PATH=$QTDIR/bin:$PATH

export MANPATH=$QTDIR/doc/man:$MANPATH　　　　　//可以不要

export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH

cd tools/qvfb

#./qvfb －width 640 －height 480 －depth 16 &

1．1．2 Executing Qt/embedded program on qvfb

#cd /linuette/host/Qt/embedded/2.3.1/

#. ./env.sh

env.sh 文件内容如下：

export QTDIR=/linuette/host/Qt/embedded/2.3.1

export QTEDIR=/linuette/host/Qt/embedded/2.3.1

export LD_LIBRARY_PATH=$QTDIR/lib:$QTEDIR/lib:$LD_LIBRARY_PATH

#cd examples/launcher

#./laucher &

1．1．3 Compiling MIZI Linux example source and Executing it on qvfb

打开两个终端，终端 1 运行 qvfb，终端 2 运行应用程序。

Terminal 1:

#cd /linuette/host/Qt/X11/2.3.1

#. ./env.sh

#cd tools/qvfb

#./qvfb －width 640 －height 480 －depth 16 &

Terminal 2:

#cd /linuette/host/example/helloworld

#. ./env.sh

env.sh 文件内容如下：

export QTDIR=/opt/host/i386-unknown-linux

export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH

#make

#./qhello －qws

如果现在想去执行另一个应用程序时，则不必关闭终端 1 的 qvfb，直接在终端 2 里先停止原先的 qhello，再启动它就可，如下：

#cd /linuette/host/example/tetris

#. ./env.sh

env.sh 文件内容如下：

export QTDIR=/opt/i386-unknown-linux

export LD_LIBRARY_PATH=/lib:/usr/lib:/$QTDIR/lib:$LD_LIBRARY_PATH

#make

#./tetris －qws

1．2 How to use tmake

1．2．1 Using tmake for x86

#cd /linuette/host/example/tetris

#. ./env.sh

env.sh 文件内容如下：

export QTDIR=/opt/i386-unknown-linux

export TMAKEPATH=/usr/lib/tmake/lib/qws/linux-x86-g++

export LD_LIBRARY_PATH=/lib:/usr/lib:/$QTDIR/lib:$LD_LIBRARY_PATH

#progen －n tetris －o tetris.pro   //自动生成*.pro

#tmake tetris.pro －o Makefile   //自动生成 Makefile

#make

再如上节所述先在另一终端设置环境变量变量执行 qvfb（如已运行，就不需要了），再在本终端执行编译的目标文件。

#./tetris －qws

1．2．2 Using tmake for arm

#cd /linuette/target/example/tetris

该目录与/linuette/host/example/tetris 目录只是 env.sh 文件不同，所以也可以直接在后者下，修改环境变量设置文件 env.sh，再执行下述操作。

#. ./env.sh

env.sh 文件内容如下：

export QTDIR=/opt/host/armv4l/armv4l-unknown-linux

export TMAKEPATH=/usr/lib/tmake/lib/qws/linux-linuette-g++

export LD_LIBRARY_PATH=/lib:/usr/lib:/$QTDIR/lib:$LD_LIBRARY_PATH

#progen －n tetris －o tetris.pro

#tmake tetris.pro －o Makefile

#make

#file tetris

tetris: ELF 32-bit LSB executable, ARM, version 1(ARM), for GNU/Linux 2.0.0, dynamically linked (uses shared libs), not stripped

将 tetris 下载到目标板上,同时目标板要下载带 GUI 的文件系统,启动后退出 Qt 服务器，运行 tetris。也可直接修改/usr/etc/rc.local 脚本，启动时运行 tetris 而不是 server。

1．1．3 出现的错误

1> 找不到/usr/lib/tmake/lib/qws/linux-x86-g++

原因是安装了 tmake-1.7-8，而不是 mizi 光盘的 tmake-1.7-3mz，另外自己下载的 tmake 也没有 tmake/lib/qws/linux-linuette-g++项。

#rpm －q tmake

tmake-1.7-8

#rpm －e tmake

#cd /linuette/RPMS

#rpm －Uvh tmake-1.7-3mz.noarch.rpm

#rpm －q tmake

tmake-1.7-3mz

2> #make 时出现如“undefined reference to ‘MzApplication::MzApplication(int &, char **)’” 等错误。原因是某些库没有链接，mizi 文档提供了三种方法，下面是我认为最方便的，因为修改后，以后编译其它应用程序时就不用再修改了。

#vi $TMAKEPATH/tmake.conf

添加“-llinuetteapp －lcustomwidget -lmzdateformat”到 TMAKE_LIBS 项，如下：

TMAKELIBS = -llinuetteapp －lcustomwidget －lmzdateformat

2. QTE 2.3.7 / Qtopia 1.7.0

国内 S3C2410 开发板上跑的大部分还是 qtopia-free-1.7.0，对应的 QTE 为 qt-embedded-2.3.7，另外还有 tmake-1.11、qt-x11-2.3.2 等。

需要将交叉编译工具安装到/usr/local/arm/，并已设置$PATH 环境变量（即在/etc/profile 里增加 pathmunge /usr/local/arm/2.95.3/bin/ ），当然也可以使用原先的/opt/host/armv4l/bin/armv4l-unknown-linux-。

2.1 Host

先到ftp.trolltech.com下载上述四个文件的源代码，存放/x86-qtopia目录后解压

#tar zxvf tmake-1.11.tar.gz

#tar zxvf qt-embedded-2.3.7.tar.gz

#tar jxvf qtopia-free-1.7.0.tar.bz2

#tar zxvf qt-x11-2.3.2.tar.gz

#mv tmake-1.11 tmake

#mv qt-2.3.7 qt

#mv qtopia-free-1.7.0 qtopia

#mv qt-2.3.2 qt-x11

2．1．1 编译 qt-x11

编译它的目的是得到 uic、moc、designer、qvfb 等所需工具。

#cd qt-x11

#export QTDIR=$PWD

#echo yes|./configure －static －no-xft －no-opengl －no-sm

#make －C src/moc

#cp src/moc/moc bin

#make －C src

#make －C tools/designer

#make －C tools/qvfb

#cp tools/qvfb/qvfb bin

#strip bin/uic bin/moc bin/designer bin/qvfb

#cd ..

#cp qt-x11/bin/* qt/bin

#rm －rf qt-x11

2．1．2 编译 qt-embedded

```
#export QTDIR=$PWD/qt
#export QPEDIR=$PWD/qtopia
#export TMAKEDIR=$PWD/tmake
#export TMAKEPATH=$TMAKEDIR/lib/qws/linux-generic-g++
#export PATH=$QTDIR/bin:$QPEDIR/bin:$TMAKEDIR/bin:$PATH
#cd qt
#make clean
#cp $QPEDIR/src/qt/qconfig-qpe.h src/tools/
#(echo yes; echo yes)|./configure －platform linux-generic-g++ -qconfig qpe －depths 16,24
#make －C src
```

2．1．3 编译运行基于Qt/Embedded的应用程序

Hello应用程序

创建、进入hello目录

```
#mkdir hello
#cd hello
#vi hello.cpp
```

编辑hello.cpp：

```
#include <qapplication.h>
#include <qpushbutton.h>
int main(int argc, char *argv[]){
    QApplication app(argc, argv);
    QPushButton hello(“hello world!”, 0);
    hello.resize(100, 30);
    app.setMainWidget(&hello);
    hello.show();
    return app.exec();
}
```

设置host运行下的环境变量

```
#vi setEnvHost
```

注：假设已经设定了TMAKEDIR,QTEDIR等变量

```
export TMAKEPATH=$TMAKEDIR/lib/qws/linux-generic-g++
export QTDIR=$QTEDIR
export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
#. ./setEnvHost
#progen －n hello －o hello.pro
#tmake －o Makefile hello.pro
#make
```

出现错误：

```
g++ -c pipe －DQWS －fno-exceptions －fno-rtti －Wall －W －O2 －DNO_DEBUG －I/x86-qtopia/qt/include －o hello.o hello.cpp
gcc －o hello hello.o －L/x86-qtopia/qt/lib －lqte
/x86-qtopia/qt/lib/libqte.so: undefined reference to ‘operator new[](unsigned)’
/x86-qtopia/qt/lib/libqte.so: undefined reference to ‘operator delete(void*)’
/x86-qtopia/qt/lib/libqte.so: undefined reference to ‘cos’
```

/x86-qtopia/qt/lib/libqte.so: undefined reference to ‘sin’

/x86-qtopia/qt/lib/libqte.so: undefined reference to ‘__cxa_pure_virtual’

/x86-qtopia/qt/lib/libqte.so: undefined reference to ‘pow’

/x86-qtopia/qt/lib/libqte.so: undefined reference to ‘operator delete[](void*)’

/x86-qtopia/qt/lib/libqte.so: undefined reference to ‘operator new(unsigned)’

collect2: ld returned 1 exit status

make: *** [hello] Error 1

原因是 Makefile 里的 LINK 项应该为 g++ 而不是 gcc，修改 tmake/lib/qws/linux-generic-g++/tmake.conf

#vi $TMAKEPATH/tmake.conf

将 TMAKE_LINK = gcc 和 TMAKE_LINK_SHLIB = gcc

分别修改成

TMAKE_LINK = g++和 TMAKE_LINK_SHLIB = g++

#tmake －o Makefile hello.pro

此时查看 Makefile 里的 LINK 项应该已经为 g++

#make

#./hello －qws　　　　　　　　　注：另一终端已运行 qvfb

2．1．4 编译运行 qtopia

qtopia 也是 QTE 的应用程序，只是它比较复杂

#cd $QPEDIR/src

#./configure －platform linux-generic-g++

#make

#cd bin

#./qpe　　　　　　　　　　　　注：另一终端已运行 qvfb

2．1．5 将应用程序添加到 qtopia

将 hello 目标文件复制到 qtopia/bin 目录下，将显示的图标 Hello.png 复制到 qtopia/pics 目录下，在 qtopia/apps/Applications 目录下增加 hello.desktop 文件，内容如下：

[Desktop Entry]

Comment=An Hello Program

Exec=hello

Icon=Hello

Type=Application

Name=HelloWorld

Exec 指出的名字必须与应用的目标二进制文件名一致；Icon 指出的是应用在 desktop 中的小图标的文件名(不含扩展名.png，如果 Hello.png 位于 pics/hello 目录下，则 Icon=/hello/Hello)；Name 指出 desktop 上显示的程序名。

另外，上述的 hello 例程是基于 QTE 单独编译后再加入到 qtopia 的，也可以和 qtopia 源码一起编译，只需稍微修改。

1> 将原来的 hello 目录复制到 qtopia/src/applications/目录下

2> 在 hello.cpp 文件中将

#include <qapplication.h>

修改成

#include <qtopia/qpeapplication.h>

3> 在 hello.pro 文件中将

CONFIG = qt warn_on release

修改成

CONFIG = qtopia warn_on release

增加指定生成目标文件的目录项

DESTDIR = $(QPEDIR)/bin

4> 按照上述的方法成生 hello.desktop 和复制图标文件，另外 hello 目录下的其它文件可以删除

2．2 Target

编译 Target 平台版本和 Host 平台版本的方法基本相同，需要将 linux-generic-g++修改成 linux-arm-g++，配置支持触摸板等。

#mkdir /arm-qtopia

#cd /arm-qtopia

#tar zxvf /x86-qtopia/tmake-1.11.tar.gz

#tar zxvf /x86-qtopia/qt-embedded-2.3.7.tar.gz

#tar jxvf /x86-qtopia/qtopia-free-1.7.0.tar.bz2

#mv tmake-1.11 tmake

#mv qt-2.3.7 qt

#mv qtopia-free-1.7.0 qtopia

2．2．1 编译 qt-x11

由于和 Host 编译方法一样，所以只需将上节的目标工具复制过来就可以。

#cp /x86-qtopia/qt/bin/* qt/bin

2．2．2 编译 qt-embedded

#export QTDIR=$PWD/qt

#export QPEDIR=$PWD/qtopia

#export TMAKEDIR=$PWD/tmake

#export TMAKEPATH=$TMAKEDIR/lib/qws/linux-arm-g++

#export PATH=$QTDIR/bin:$QPEDIR/bin:$TMAKEDIR/bin:$PATH

#cd qt

#make clean

#vi $QPEDIR/src/qt/qconfig-qpe.h

增加支持触摸屏的宏定义：

#define QT_QWS_IPAQ

#define QT_QWS_IPAQ_RAM

#cp $QPEDIR/src/qt/qconfig-qpe.h src/tools/

#(echo yes; echo yes)|./configure －platform linux-arm-g++ -qconfig qpe -no-qvfb －depths 16,24

#make －C src

2．2．3 编译运行基于 Qt/Embedded 的应用程序

使用上一节的 Hello 应用程序

#cd /x86-qtopia/hello

重新设置环境变量，如果还在 2.2.2 节的终端下，就不用设置

#vi setEnvArm

export QTDIR=/arm-qtopia/qt

export TMAKEDIR=/arm-qtopia/tmake

export TMAKEPATH=$TMAKEDIR/lib/qws/linux-arm-g++

export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH

#. ./setEnvArm

重新生成Makefile，由于*.pro与平台无关，所以不用重新生成

#tmake －o Makefile hello.pro

#make

下载目标板运行

将/arm-qtopia/qt/lib目录下的文件下载到目标板某个目录，设置相应的环境变量，再下载hello运行即可。

另外要注意：libqte.so.2、libqte.so.2.3 等为链接文件，如果用 ztelnet 直接发送（如 sz libqte.so.2)，会将所链接的libqte.so.2.3.7发送过去，此时的目标板上libqte.so.2、libqte.so.2.3都已不是链接而是实际的文件，flash会不够用。

2．2．4 qtopia

#cd $QPEDIR/src

#vi libraries/qtopia/custom.h

增加触屏校准程序的宏定义：#define QPE_NEED_CALIBRATION

#./configure －platform linux-arm-g++

#make

该版本的Makefile文件里没有make install?，只好手动打包了。在linuette的root_english基础上

#cd /linuette/target/box/root_dir/root_english/usr

将qtopia替换linuette的桌面系统

#rm －rf /linuette/target/box/root_dir/root_english/usr/linuette

#mkdir /linuette/target/box/root_dir/root_english/usr/qtopia

#cd /arm-qtopia/qtopia

#cp －ir apps bin etc help i18n lib pics plugins services sounds /linuette/target/box/root_english/usr/qtopia

将qt-embedded-2.3.7的库替换linuette的2.3.1库文件

#cd /linuette/target/box/root_dir/root_english/usr/qt/lib

#rm －rf *

#cp /arm-qtopia/qt/lib/libqte.so.2.3.7 ./

#ln －s libqte.so.2.3.7 libqte.so

#ln －s libqte.so.2.3.7 libqte.so.2

#ln －s libqte.so.2.3.7 libqte.so.2.3

#ln －s ../etc/fonts fonts

修改环境变量

#vi ../../etc/profile

增加或修改

QTDIR=/usr/qt

QPEDIR=/usr/qtopia

LD_LIBRARY_PATH=/usr/qt/lib:/usr/qtopia/lib:/lib:/usr/lib

桌面启动

#vi ../../etc/rc.local

屏蔽最后三行后增加：

qpe

2．3 增加JPEG库

上述无论时Host还是Target在执行Image Viewer浏览图片时会显示装载失败，原因是在编译时没有添加jpeg库等支持。

下述为编译Host版本（Target版本方法类似）

先要下载jpegsrc.v6b.tar.gz、libpng-1.2.14.tar.bz2解压

#tar zxvf jpegsrc.v6b.tar.gz －C /home

#tar jxvf libpng-1.2.14.tar.bz2 －C /home

#cd /home

#mv jpeg-6b jpeg

#mv libpng-1.2.14 libpng

创建两个文件夹用于存放编译后的库及头文件

#mkdir lib

#mkdir include

编译生成jpeg库

#cd jpeg

#./configure －enable-shared

#make

#cp .libs/libjpeg.so* ../lib

#cp *.h ../include

编译生成libpng

#cd /home/libpng

#cp scripts/makefile.linux ./Makefile

#make

#cp libpng12.so* /home/lib

#cp libpng12.so /home/lib/libpng.so

#cp *.h /home/include/

重新配置编译Qt/E

#cd $QTDIR　　　　　　　　　注：假设已设置好环境变量

#vi $TMAKEPATH/tmake.conf

将此行

QMAKE_LIBS_QT = -lqte

修改为

QMAKE_LIBS_QT = -lqte -lpng －ljpeg

#(echo yes; echo yes)|./configure －platform linux-generic-g++ -qconfig qpe －depths 16,24 -system-jpeg -system-libpng -gif -I/home/include -L/home/lib －lpng －ljpeg

#make －C src

重新编译qtopia

#cd $QPEDIR/src

#./configure －platform linux-generic-g++

#make

#cd bin
#./qpe　　　　　　　　　　　注：另一终端已运行 qvfb

3. QTE 2.3.12 / Qtopia PDA 2.2.0

从 Qtopia 2.2.0 开始，它包含了 QTE 的源代码及 tmake 等，所以不需要另外下载。

除官网外，还可从下述下载：

http://www.qtopia.org.cn/ftp/mirror/ftp.trolltech.com/qtopia/source/qtopia-free-src-2.2.0.tar.gz

3．1 Qtopia PDA 2.2.0(GPL) Host 安装指南

编译：

#makedir /x86_qtopia2.2.0
#tar zxvf qtopia-free-src-2.2.0.tar.gz －C /x86_qtopia2.2.0/
#cd /x86_qtopia2.2.0/qtopia-free-src-2.2.0
#export PATH=$PWD/tmake/bin:$PATH
#echo "yes"|./configure -qte no-keypad -qpe pda
#make

Make 时可能出错：make clean 再 make

#make install

运行：

#cd qtopia-free-2.2.0/qtopia/image/opt/Qtopia
#mkdir demohome
#cd ../../../bin
#./startdemo －home qtopia/image/opt/Qtopia/demohome -sound system

错误：

Detected Qtopia edition ‘pda’ in /home/qtopia-free-2.2.0/qtopia
Using ‘system’ sound daemon
Starting qvfb …
QVFB 0 is already running!
Starting Qtopia pda …
Can’t open framebuffer device /dev/fb0
Driver cannot connect
Stopping qvfb…

在官方文档（http://trolltech.com/developer/knowledgebase/661）中找到下述问题：

How do I find out the properties of my Linux Frame Buffer?
Entry number: 661 - How do I find out the properties of my Linux Frame Buffer?
Answer:
Use the Linux command: fbset, often found in /usr/sbin/.

输入 fbset 测试命令后，出现下述问题：

[root@localhost bin]# fbset

open /dev/fb0: No such device

但用 ls 命令能找到/dev/fb0 设备

[root@localhost bin]# ls /dev/fb0 －l

crw------- 1 root root 29, 0 Jan 30 2003 /dev/fb0

原因是 framebuffer 没有打开，打开方法：在/etc/grub.conf（或/boot/grub/menu.lst）文件的 kernel 一行后面加入 vga=791 fb=on，然后重新启动系统，启动的开始会出现一个企鹅标志。另外一种打开方法是，在 GRUB 启动引导菜单中 windows XP 和 Red Hat Linux(2.4.20-8)两项中使用键头键选中 linux 系统，不按[Enter]，而按[E]键进入菜单项目编辑器，再使用键头键选中 kernel 项，也按[E]键进行编辑，在行的后面输入 vga=791 fb=on 后按[Enter]，最后按[b]键执行命令，并引导操作系统。

错误：修改后不起作用，刚启动时也没有出现企鹅标志，原因是将 fb=on 写成了 fb:on。另外个别电脑将 vga=791 时，启动时显示器不显示，应该是显卡及驱动本身问题。

3．2 Target 交叉编译

除 qtopia-free-src-2.2.0.tar.gz 外，另需要 e2fsprogs-1.35.tar.gz、jpegsrc.v6b.tar.gz、libpng-1.2.14.tar.bz2。

3．2．1 编译步骤

1> 复制上节 Host 版本下的 uic 工具

#mkdir /arm_qtopia2.2.0

#tar zxvf qtopia-free-src-2.2.0.tar.gz －C /arm_qtopia2.2.0

#cd /arm_qtopia2.2.0/qtopia-free-2.2.0

#cp /x86_qtopia2.2.0/qtopia-free-2.2.0/qt2/bin/uic ./qt2/bin/

2> 编译相关库

#cd ../

#tar zxvf /qt_tools/e2fsprogs-1.35.tar.gz

#tar zxvf /qt_tools/jpegsrc.v6b.tar.gz

#tar jxvf /qt_tools/libpng-1.2.14.tar.bz2

#mv e2fsprogs-1.35 e2fs

#mv jpeg-6b jpeg

#mv libpng-1.2.14 libpng

#mkdir lib

#mkdir include

编译 e2fs

#cd e2fs

#./configure --host=arm-linux --enable-elf-shlibs --with-cc=arm-linux-gcc --with-linker=arm-linux-ld --prefix=/usr/local/arm/2.95.3/arm-linux

#make

#cp lib/libuuid.so* ../lib

编译 jpeg

#cd ../jpeg

#./configure －enable-shared

#vi Makefile

修改:

```
    CC= arm-linux-gcc
    AR= arm-linux-ar rc
    AR2=arm-linux-ranlib
#make
#cp .libs/libjpeg.so* ../lib
cp *.h ../include
```

编译 libpng

```
#cd ../libpng
#cp  scripts/makefile.linux  ./Makefile
#vi Makefile
```

修改:

```
    AR_RC=arm-linux-ar rc
    CC=arm-linux-gcc
    RANLIB=arm-linux-ranlib
    prefix=/usr/local/arm/2.95.3/arm-linux
#make
#cp libpng12.so* ../lib/
#cp libpng12.so ../lib/libpng.so
#cp *.h ../include/
```

3> 编译 Qtopia PDA

建立 PDA 安装目录

```
#mkdir ../qtopia
#cd ../qtopia-free-2.2.0
```

修改文件

```
#vi qtopia/mkspecs/qws/linux-arm-g++/qmake.conf
```

将此行

```
    QMAKE_LIBS_QT = -lqte
```

修改为

```
    QMAKE_LIBS_QT = -lqte -lpng -luuid - ljpeg
#vi qtopia/src/qt/qconfig-qpe.h
```

增加触摸屏支持的宏定义：

```
#define QT_QWS_IPAQ
#define QT_QWS_IPAQ_RAW
#cp qtopia/src/qt/qconfig-qpe.h qt2/src/tools
#cd qtopia/src/libraries/qtopia
#cp custom-linux-ipaq-g++.cpp custom-linux-arm-g++.cpp
#cp custom-linux-ipaq-g++.h custom-linux-arm-g++.h
#echo yes | ./configure - qte “-embedded -xplatform linux-arm-g++ -qconfig qpe -no-qvfb -depths 16,24 -system-jpeg -system-libpng -system-zlib -gif -thread -no-xft -release -I/home/arm/include -L/home/arm/lib - lpng -lz -luuid - ljpeg” -qpe “-xplatform linux-arm-g++ -edition pda -displaysize 640x480 -I/home/arm/include -L/home/arm/lib -prefix=/home/nfs/qtopia”
#. ./setQpeEnv
#make
#make install
```

此进将在/x86-qtopia/qtopia目录下生成在目标板运行所需的一切文件

3．2．2 下载到目标板运行

#mkdir /image/qtopia_2.2.0

#tar jxvf /linuette/target/box/root_dir/root_english.tar.bz2 －C /image/qtopia_2.2.0/

#cd /image/qtopia_2.2.0/root_english/usr

#rm －rf linuette

#cp －ir /arm-qtopia/qtopia ./

#mkdir share

#cp －ir /arm-qtopia/qtopia-free-2.2.0/qtopia/zoneinfo ./share

#mkdir ../Documents //用于存放自己的（如mp3,image等）文件

#vi ./etc/profile

在PATH行的最后增加:/usr/qtopia/bin

在LD_LIBRARY_PATH行的=后插入/usr/qtopia/lib : 注：必须放在最前面，否则会出错

QTDIR行改为QTDIR=/usr/qtopia

增加行QPEDIR=/usr/qtopia

export行最后增加QPEDIR项

#vi ./etc/rc.local

最后三行屏蔽掉，增加pqe启动命令，如：

#cp /qt/etc/pointercal /etc

#calibrate －qws

#server －qws

qpe

3．2．2 出现的错误

1> make qte 时出现 kernel/qjpegio.cpp:60: jpeglib.h: No such file or directory 等错误，原因是当时没有将jpeg下的头文件复制过去，而缺少jpeg所需的头文件。

#cp /arm-qtopia/jpeg/*.h /arm-qtopia/include

当qte没有错误时，将在qtopia/lib目录下产生libqte.so.2.3.12

2> make qpe时出现错误：

In file included from videocaptureview.cpp:54:

/usr/local/arm/2.95.3/lib/gcc-lib/arm-linux/2.95.3/../../../../arm-linux/sys-include/linux/videodev.h:5: linux/version.h: No such file or directory

#cp /linuette/target/box/kernel/include/linux/version.h /usr/local/arm/2.95.3/arm-linux/include/linux/

或在/usr/local/arm/2.95.3/arm-linux/include/linux/ videodev.h 文件和/usr/local/arm/2.95.3/arm-linux/sys-include/linux/ videodev.h 文件里的#include <linux/version.h>行屏蔽掉

3> Make qpe时出现错误：

/home/nfs/qtopia-free-2.2.0/qtopia/lib/libqte.so: undefined reference to ‘__fixsfsi’

/home/nfs/qtopia-free-2.2.0/qtopia/lib/libqte.so: undefined reference to ‘__subsf3’

/home/nfs/qtopia-free-2.2.0/qtopia/lib/libqte.so: undefined reference to ‘__floatsidf’

……

原因是缺少jpeg所需的库文件

#cp /arm-qtopia/jpeg/.libs/libjpeg.so* /arm-qtopia/lib

#make clean

#make

4> 运行 qpe 时出现装载共享库错误

#./qpe

./qpe: error while loading shared libraries: libpng12.so.0: cannot load shared object file: No such file or directory //原因是没有将/arm-qtopia/lib 下的库文件复制到/arm-qtopia/qtopia/lib

#cp /arm-qtopia/lib/* /arm-qtopia/qtopia/lib

5> 不能写 qpe_new.conf

could not open for writing `/Settings/qpe_new.conf'

QCopChannel::send: Must construct a QApplication before using QCopChannel

原因是：不能创建/Settings/qpe_new.conf，因为系统根目录为只读文件系统，要换成 yaffs 等可写

6> 不能打开/dev/fb0

#./qpe

Can’t open framebuffer device /dev/fb0

Can’t open framebuffer device /dev/fb0

driver cannot connect

原因是：没有/dev/fb0 设备，只有/dev/fb/0 为 framebuffer 设备，所以要创建/dev/fb0 到/dev/fb/0 的链接，如下：

#ln –s /dev/fb/0 /dev/fb0

7> 触摸屏屏问题

#./qpe

Mouse type Auto unsupported

#ln –s /dev/touchscreen/0raw /dev/h3600_tsraw

#export QWS_MOUSE_PROTO=/dev/h3600_tsraw //注：后来实验证明这行不用也可以

#./qpe

Mouse type /dev/h3600_tsraw unsupported

在 qtopia-free-2.2.0/qt2/src/tools/qconfig-qpe.h 中增加

#define QT_QWS_IPAQ

#define QT_QWS_IPAQ_RAW

8> 找不到 zoneinfo

#./qpe

Madvise of shared memory: Function not implemented

Madvise error with marking shared memory pages as not needed

Warning: Unable to open /usr/share/zoneinfo/zone.tab

Warning: Timezone data must be installed at /usr/share/zoneinfo/

Warning: TimeZone::data Can’t create a valid data object for ‘@H!q@’

警告：

找不到 zoneinfo

#mkdir /usr/share

#cp –ir /arm-qtopia2.2.0/qtopia-free-2.2.0/qtopia/etc/zoneinfo /usr/share

4. linuette 的 root、root_english、usr 比较

由于要将 qtopia 添加到 linuette 的文件系统中，那么怎么添加呢？添加到哪个比较合适，什么文件及目录可以删除？所以我觉得很有必要了解一下 linuette 提供的几个文件系统目录。

/根目录：root 和 root_english 相同，包含：

```
bin  dev  etc  lib  linuette  linuxrc  mnt  proc  qt  sbin  tmp  usr  var
# ls
bin   lib       mnt       sbin      var
dev   linuette  proc      tmp
etc   linuxrc   qt        usr
```

如下为 root_english 系统下所例：

```
# ls -l
drwxr-xr-x        1 0      0              512 Jun 15  2003  bin
drwxr-xr-x        1 0      0                0 Jan  1  1970  dev
drwxr-xr-x        1 0      0                0 Nov 27  1990  etc
drwxr-xr-x        1 0      0              512 Mar 28 10:31  lib
lrwxrwxrwx        1 0      0               13 Jun 18  2007  linuette -> /etc/linuette
-rwxr-xr-x        1 0      0              274 Jan  1  1970  linuxrc
drwxr-xr-x        1 0      0              512 Mar 28 10:31  mnt
dr-xr-xr-x        220      0                0 Nov 27  1990  proc
lrwxrwxrwx        1 0      0                7 Jun 18  2007  qt -> /usr/qt
drwxr-xr-x        1 0      0              512 Mar 21 04:51  sbin
lrwxrwxrwx        1 0      0                8 Jun 18  2007  tmp -> /etc/tmp
drwxr-xr-x        1 0      0              512 Mar 28 10:31  usr
lrwxrwxrwx        1 0      0                8 Jun 18  2007  var -> /etc/var
```

root 与 root_english 唯一不同处为 linuette 的链接/usr/linuette，如下：

```
lrwxrwxrwx        1 0      0               13 Jun 18  2007  linuette -> /usr/linuette
```

bin 目录下 root_englis 比 root 目录多 irdadump、obex_test

dev 目录都为空，用于启动时被 devfs 挂载

etc 目录都为空，在启动脚本（linuxrc）中，将被 mount ramfs（因为它必须为可写），然后将/mnt/etc 目录下的内容复制过来

lib 目录也相同，其中 modules 也都被链接到/usr/lib/modules

linuxrc 脚本的内容也相同，都是先将/etc 目录 mount 成可写的 ramfs 系统，然后将/mnt/etc 下内容复制过来，然后 exec /sbin/init

mnt 目录也都有 etc、ext1、ext2、ext3 四个目录，除 etc 外都为空。etc 目录中，root_english 除了多 linuette 目录外，其它也都一样。etc/init.d/rcS 脚的内容也都为：

```
# cat init.d/rcS
#!/bin/sh
/bin/mount -a
exec /usr/etc/rc.local
```

其中，rc.local 为启动 QPE 的脚本

linuette 内容为：

```
# ls linuette -l
lrwxrwxrwx    1 0         0              17 Dec   4  2030      bin -> /usr/linuette/bin
drwxr-xr-x    1 0         0               0 Jan   1  1970      config
lrwxrwxrwx    1 0         0              17 Dec   4  2030      lib -> /usr/linuette/lib
lrwxrwxrwx    1 0         0              20 Dec   4  2030      locale                     ->
```

/usr/linuette/locale

```
 drwxr-xr-x    1 0        0                  0 Mar 21 04:51     share
 drwxr-xr-x    1 0        0                  0 Dec 20 12:12     start
```

proc 目录也都为空

qt 也都被链接到/usr/qt

sbin 目录内容也都相同，sbin/init 也都被链接到/bin/busybox

tmp、var 也都被链接到/etc/目录下的 tmp、var，都为可写目录

usr 目录，root 系统为空，可用于另挂载文件系统使用；而 root_english 的 usr 根目录和 usr 系统的根目录内容相同，如下：

#ls usr

bin   etc   lib   linuette   qt   sample   sbin

其中，bin、etc、lib、qt、sbin 相同，而 usr 系统的 sample 为空，root_english 的 usr/sample 存放了 jpe 图片和 mpg、mp3 多媒体文件，用于测试。usr 系统下的 linuette 目录如下：

bin   config   lib   locale   share   start

而 root_english/usr/linuette 只有 bin、lib 目录，内容基本相同，bin 存放着 QPE 的应用程序，lib 存放 QPE 的库文件。其实在 root_englis 系统下根目录的 linuette 下是向 usr 系统那样链接到该目录（/usr/linuette），而是/etc/linuette（即/mnt/etc/linuette），它的内容和 linuette 基本一样，如下：

bin   config   lib   locale   share   start

其中 bin、lib 是分别链接到/usr/linuette 目录下的 bin 和 lib。

后记

在技术上，我总是很贪图，当初掌握 51 后，马上就想着功能更强的 51，后来 PIC，甚至还想过 AVR、DSP，想过做做模拟、画画 PCB 等等。掌握 ARM 后，马上就想着 linux，而且把目标定位在能胜任底层驱动、应用、GUI 和网络等任何位置的编程调试，我甚至想过改行去做几年 VC 等的 PC 机程序员。嗨！真希望有谁能拿起砖头往我头上重重的砸几下，然后告诉我这错误的道理。还好我明白这里的很多东西是相通的，不用特意的花时间精力去学习，来克制自己的这种欲望，也时常告诫自己要精通一门，技多不养家等等。虽然如此，我已经决定在业余时间玩玩 wince，因为我深刻的体会到目前在方案选择上的局限性!因为没有掌握和深入使用 wince，自然就不会切身体会到它的优缺点，那么在项目中又如何能够正确的判断出到底是 wince 还是 linux 更适合？即使很明显是 wince 适合，也找到了合适的工程师来实现，万一在开发过程中出现问题怎么办？在旁边干着急多不爽！哈哈。。。。

阿南：ccn422@hotmail.com 欢迎交流！